



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍并不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战

非卖品，仅供非商业用途或交流学习使用

从前端、网络、CDN到后端、大促的全链路性能优化详解

周清明 张荣华 张新兵 著

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战

从前端、网络、CDN到后端、大促的全链路性能优化详解

周清明 张荣华 张新兵 著

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

性能是大型网站的一个要素，影响因素非常多。本书由三位熟悉不同领域性能优化的技术专家打造，从大型网站的整体体系出发，讲述大型网站性能优化的全链路实践过程，包括核心原理、常见策略与实战案例。具体内容包括：基于用户体验的性能优化要素、前端性能优化实战、网站性能分析、服务端性能优化、TCP优化、DNS优化、CDN优化、大型网站性能监控体系、大型网站容量评估、高性能系统架构模式、大促保障体系、数据分析驱动性能优化。

本书的初衷就是将实践经验分享给读者，展示性能优化相关知识的全貌。书中的很多性能优化方法和策略都是作者从实践中总结出来的，实用性非常强。本书既可供入门者了解大型网站性能优化所有的相关技术，以及解决问题的思路和方法，也可供业界同行参考，给日常工作带来启发。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解 / 周清明，张荣华，张新兵著. —北京：电子工业出版社，2019.1
ISBN 978-7-121-35002-3

I. ①大… II. ①周… ②张… ③张… III. ①网站—开发 IV. ①TP393.092

中国版本图书馆CIP数据核字（2018）第207734号

责任编辑：董英
印刷：三河市双峰印刷装订有限公司
装订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036

开本：787×960 1/16 印张：21 字数：455千字
版次：2019年1月第1版
印次：2019年1月第1次印刷
定价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888、88258888。

质量投诉与维权电话：010-88258888，盗版侵权举报请发邮件至：dbqq@phei.com.cn。
本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

人生是一场修炼的过程，不断地突破自己的舒适圈，不断地和内心的懒惰做斗争，咬牙做下来，才能体会到这件事情的不易，轻易得到的总不会太珍惜。我们处于飞速发展的时代，同时我们也处于浮躁的时代，这个时代造就了很多英雄，但也很容易在其中迷失自己。在迷失的时候，给自己定一个目标，只有不断地学习和努力，同时摆正自己的心态，明确自己想要的东西，才能处在浪潮之巅，在时代的浪潮里留下自己的痕迹。希望本书也能留下一丝痕迹。

关于作者

本书的作者是正在实践中逐步成长起来的架构师，在项目的实践中，全力以赴地解决各种难题。笔者2008年认识荣华，2011年认识新兵，他们给了笔者很多支持。特别是2008年在Cisco工作的过程中，荣华给了笔者很多支持，笔者跟他学到了很多。当年JavaEye非常盛行的时候，荣华（以笔名ahuaxuan）在JavaEye论坛名气很大。笔者在加入Cisco之后，有幸与他成为同事，他教会了笔者很多东西。新兵负责过前端性能优化部分，他在前端性能方面有比较深入的研究，我们一起合作过多个性能优化的项目，并且在实践过程中取得了良好的效果。大家因为共同的爱好和兴趣而聚在一起，在策划这本书时，我们很快形成共识，大家因为志同道合而相聚，都知道这本书的意义。作为一个工程师或者专家很难做到的，本书主要突出全面性和实战性，是目前笔者认为的最全面的关于性能优化的书籍，本书试图打造端到端的优化理论和实战体系。

本书的体系是非常广的，可能只有很少数的人对每个部分都比较精通，碰巧的是我们3个人，是熟悉不同领域的工程师，这样才可以各自发挥特长，让本书更有味道，也更有广度。本书是市面上第一本从前端到后端，从CDN、DNS到TCP、到机房、大促全链路的关于性能优化的书籍，能够通晓这些知识，是一个工程师或者专家很难做到的，本书主要突出全面性和实战性，是目前笔者认为的最全面的关于性能优化的书籍，本书试图打造端到端的优化理论和实战体系。

本书的主要结构

本书以大型网站性能优化实战为主题，讲述了性能优化的基本理论和实践策略。

第1章介绍影响用户体验的几个关键要素，重点讲解白屏、首屏及页面整体加载的过程，针对它们提出了比较系统的优化策略和常见方案。

• IV •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• V •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• VI •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• VII •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• VIII •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• IX •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• X •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

第3章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的这些关键性能指标，从而使我们能够快速对网站性能问题进行分析并做出优化。

第4章讲述服务器端性能优化理论体系和实战，包括服务器端的性能优化方法和常见的优化策略。重点包括QPS的优化、同步模型与异步模型对性能的影响、数据结构对性能的影响、算法设计不合理带来的性能问题，以及一个综合案例。

第5章阐述TCP优化，介绍了TCP的基本原理，并对TCP近几年的发展做了介绍，最后介绍了一个项目中的TCP优化实战案例。

第6章讲述DNS的一些优化方案，主要从DNS的基本原理出发，结合跨境DNS部署和实战经验，介绍了DNS优化的历程。

第7章介绍CDN的优化实践，重点包括CDN的工作原理、CDN优化的常见策略，同时介绍了大量的优化案例，最后总结了CDN的优化原则。

第8章主要讲述大型网站的性能监控需求和监控指标，以及如何实现监控，揭开大型网站性能监控体系的面纱。

第9章主要介绍大型网站如何进行容量评估，重点包括单机峰值QPS的测算、大型网站常用的容量评估方法。

第10章高性能系统架构模式，主要从宏观角度来看性能优化。好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。好的高阶架构比局部优化带来的效果要大得多，如同人们经常说的，格局决定高度。

第11章重点介绍大促的整体方案和细节工作，轴以案例，加深读者的印象。大促保障是性能优化的重要力量，由于大促保障本身是成体系的，除了性能优化，还介绍了稳定性保障和资金安全保障的内容。

第12章数据分析驱动性能优化，从数据视角介绍如何进行性能优化，重点包括性能优化相关的分析原理与方法，以及如何在实践中使用数据分析来进行大型网站的优化。

周清明
2018年11月

• XI •

非卖品，仅供非商业用途或交流学习使用

大型网站性能优化实战：从前端、网络、CDN到后端、大促的全链路性能优化详解

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35002

序言

第2章主要介绍前端性能优化实战，包括延迟渲染和SEO页面的优化。重点讲解了加快页面渲染速度的基本思路及具体的实践解决方案，以及SEO页面的优化思路 and 解决方案。

</



目 录

| | |
|---------------------------|----|
| 第 1 章 基于用户体验的性能优化要素 | 1 |
| 1.1 页面用户体验的要素介绍 | 1 |
| 1.2 白屏时间 | 3 |
| 1.2.1 白屏时间的重要性 | 3 |
| 1.2.2 白屏过程详解 | 4 |
| 1.3 首屏时间 | 10 |
| 1.3.1 首屏时间的定义 | 10 |
| 1.3.2 首屏时间的重要性 | 11 |
| 1.4 页面整体加载完成 | 15 |
| 第 2 章 前端性能优化实战 | 16 |
| 2.1 延迟渲染 | 16 |
| 2.1.1 挑战和困难 | 17 |
| 2.1.2 解决方案 | 17 |
| 2.2 SEO Ajax | 20 |
| 2.2.1 挑战和困难 | 21 |
| 2.2.2 解决方案 | 21 |
| 第 3 章 网站性能分析 | 24 |
| 3.1 快速了解网站性能 | 24 |
| 3.1.1 使用 YSlow 进行性能分析 | 24 |
| 3.1.2 使用 PageSpeed 进行性能分析 | 25 |





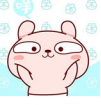
| | | |
|-------|-------------------------------|----|
| 3.1.3 | 使用 WebPagetest 进行性能分析 | 27 |
| 3.2 | 真实用户前端性能监控 | 29 |
| 3.2.1 | 真实用户前端性能数据采集 | 29 |
| 3.2.2 | 数据采集可行性分析 | 30 |
| 第 4 章 | 服务端性能优化 | 36 |
| 4.1 | 最大 QPS 推算及验证 | 36 |
| 4.1.1 | RT | 37 |
| 4.1.2 | 单线程 QPS | 38 |
| 4.1.3 | 最佳线程数 | 38 |
| 4.1.4 | 最大 QPS | 39 |
| 4.1.5 | 实验数据验证公式 | 44 |
| 4.1.6 | 压力测试最佳线程数和 QPS 的临界点 | 47 |
| 4.2 | 同步模型与异步模型 | 49 |
| 4.2.1 | 同步模型 | 49 |
| 4.2.2 | 异步模型 | 50 |
| 4.2.3 | 为什么异步模型需要的线程数少 | 58 |
| 4.2.4 | 两个模型的对比及异步模型适用场景 | 59 |
| 4.2.5 | 小结 | 60 |
| 4.3 | 数据结构对性能的影响 | 61 |
| 4.3.1 | HashMap 的问题 | 61 |
| 4.3.2 | HashMap 的结构 | 62 |
| 4.3.3 | 碰撞 | 64 |
| 4.3.4 | Hash 算法 | 65 |
| 4.3.5 | 题外话：ConcurrentHashMap 中的 Hash | 72 |
| 4.3.6 | HashMap 综述 | 73 |
| 4.3.7 | 均摊 | 74 |
| 4.4 | 算法设计不合理带来的性能问题 | 78 |
| 4.4.1 | 某应用 A 的现象 | 78 |
| 4.4.2 | 某应用 B 的现象 | 78 |
| 4.4.3 | 分析 | 79 |
| 4.4.4 | 方案 | 81 |





| | | |
|-------|----------------------------------|-----|
| 4.4.5 | 验证 | 82 |
| 4.4.6 | 小结 | 86 |
| 4.5 | 综合案例：电商活动页面性能优化 | 86 |
| 4.5.1 | 第一轮：通过 APC 使 QPS 提高近 3 倍 | 86 |
| 4.5.2 | 第二轮：解决消耗 CPU 资源大户 Gzip | 92 |
| 4.5.3 | 小结 | 105 |
| 第 5 章 | TCP 优化 | 107 |
| 5.1 | TCP 传输原理 | 108 |
| 5.1.1 | TCP 传输的简要说明 | 108 |
| 5.1.2 | 滑动窗口——接收端流量控制 | 108 |
| 5.1.3 | 拥塞窗口——发送端流量控制 | 109 |
| 5.1.4 | 传统 TCP 拥塞控制问题 | 110 |
| 5.2 | Linux 内核升级中的 TCP 优化技术 | 110 |
| 5.2.1 | 调整接收窗口 | 111 |
| 5.2.2 | 初始拥塞窗口调整（Linux 2.6.38 开始支持） | 111 |
| 5.2.3 | Early Retransmit（Linux 3.5 开始支持） | 112 |
| 5.2.4 | 初始 RTO 调整（Linux 2.6.18 开始支持） | 114 |
| 5.2.5 | TFO | 114 |
| 5.2.6 | TSO | 115 |
| 5.3 | TIME_WAIT 问题案例分析 | 116 |
| 5.3.1 | 问题现象 | 116 |
| 5.3.2 | 问题分析 | 117 |
| 5.3.3 | 问题初步解决 | 118 |
| 5.3.4 | 问题再分析 | 118 |
| 5.3.5 | 问题后记 | 119 |
| 5.4 | 总结 | 119 |
| 第 6 章 | DNS 优化 | 120 |
| 6.1 | DNS 基本原理 | 121 |
| 6.1.1 | DNS 的一些关键术语 | 121 |
| 6.1.2 | DNS 查询过程 | 122 |
| 6.1.3 | NS 选择策略和机制 | 124 |





| | | |
|-------|----------------------------|-----|
| 6.1.4 | DNS 扩展协议 EDNS | 125 |
| 6.1.5 | 常用 DNS 相关命令 | 126 |
| 6.2 | 实战案例：超远距离 DNS 性能问题分析和优化 | 130 |
| 6.2.1 | 现象描述 | 130 |
| 6.2.2 | DNS Lookup 耗时长的问题分析 | 131 |
| 6.2.3 | DNS 解析性能解决方案 | 133 |
| 6.3 | 总结 | 136 |
| 第 7 章 | CDN 优化 | 138 |
| 7.1 | CDN 优化概述 | 138 |
| 7.2 | CDN 的相关术语 | 140 |
| 7.3 | 从应用看 CDN 的基本原理 | 141 |
| 7.3.1 | CDN 基本架构 | 141 |
| 7.3.2 | CDN 全局调度 | 141 |
| 7.3.3 | CDN 基本调度方式 | 142 |
| 7.3.4 | CDN 加速的基本实施流程 | 145 |
| 7.4 | CDN 优化常见策略 | 146 |
| 7.4.1 | 静态化缓存优化 | 146 |
| 7.4.2 | 动态内容静态边缘化 | 147 |
| 7.4.3 | 动态加速优化 | 150 |
| 7.4.4 | 用户序列优化原理 | 153 |
| 7.4.5 | 域名合并优化 | 153 |
| 7.4.6 | 多级缓存架构优化 | 154 |
| 7.4.7 | 301、302 跳转边缘化访问和多终端边缘化判断 | 154 |
| 7.5 | CDN 优化实战 | 155 |
| 7.5.1 | CDN 的不合理架构造成 304 请求耗时长优化实战 | 155 |
| 7.5.2 | 静态资源命中率优化实战 | 159 |
| 7.5.3 | CDN 动态加速优化实战 | 164 |
| 7.5.4 | CDN 静态化的问题和优化实战 | 171 |
| 7.5.5 | CDN 调度优化实战 | 178 |
| 7.6 | 总结 | 179 |



| | |
|--|-----|
| 第 8 章 大型网站性能监控体系 | 182 |
| 8.1 监控设计 | 183 |
| 8.1.1 应用监控存在的问题 | 183 |
| 8.1.2 从问题排查思路看监控的设计 | 183 |
| 8.1.3 监控的设计步骤 | 184 |
| 8.1.4 监控常见法则总结 | 187 |
| 8.2 大型网站性能监控体系设计目标和原则 | 188 |
| 8.2.1 准确性 | 188 |
| 8.2.2 完整性 | 189 |
| 8.2.3 实时性 | 189 |
| 8.2.4 细分化 | 189 |
| 8.2.5 聚合化 | 189 |
| 8.2.6 图表化 | 190 |
| 8.2.7 可追溯 | 190 |
| 8.3 性能指标和监控项及实现 | 190 |
| 8.4 性能监控的关键指标 | 194 |
| 8.4.1 应用监控 | 194 |
| 8.4.2 系统监控 | 196 |
| 8.5 常用监控命令详解 | 201 |
| 第 9 章 大型网站容量评估 | 205 |
| 9.1 容量评估概述 | 205 |
| 9.2 容量评估的特点 | 206 |
| 9.3 单机峰值 QPS 的测算 | 206 |
| 9.3.1 单机测算方法 | 207 |
| 9.3.2 两种常用的引流压力测试方法 | 207 |
| 9.3.3 引流压力测试停止时间的判断 | 208 |
| 9.3.4 如何避免单机压力测试出现问题 | 209 |
| 9.4 大型网站常用的容量评估方法 | 210 |
| 9.4.1 二八原则评估法——新业务评估的基本方法 | 210 |
| 9.4.2 有历史数据参考的容量评估——GMV 线性比例评估法和 GMV 转化评估法 | 210 |
| 9.4.3 流量占比评估法 | 215 |



| | |
|---|-----|
| 9.5 总结 | 216 |
| 第 10 章 高性能系统架构模式 | 218 |
| 10.1 无状态架构 | 219 |
| 10.1.1 解决方案一——Session 复制 | 219 |
| 10.1.2 解决方案二——Session Sticky | 220 |
| 10.1.3 解决方案三——Session 集中式存储 | 220 |
| 10.1.4 解决方案四——基于浏览器 Cookie 的无状态架构 | 222 |
| 10.2 基于负载均衡器的水平扩展架构 | 222 |
| 10.3 基于 DNS 的负载均衡 | 224 |
| 10.4 读写分离架构 | 224 |
| 10.5 基于数据水平切分的水平扩展架构 | 225 |
| 10.6 缓存架构 | 228 |
| 10.6.1 缓存的基本属性 | 229 |
| 10.6.2 缓存的分类 | 229 |
| 10.6.3 缓存使用常见的问题和误区 | 230 |
| 10.6.4 缓存使用场景 | 231 |
| 10.6.5 缓存使用规范和原则 | 232 |
| 10.7 近端架构 | 233 |
| 10.8 异步化架构 | 234 |
| 10.9 排队缓冲架构 | 235 |
| 10.10 多机房架构 | 236 |
| 10.10.1 同城架构 | 236 |
| 10.10.2 异地架构 | 238 |
| 10.11 基于服务的可扩展架构 | 240 |
| 10.12 日结架构 | 242 |
| 10.13 热点避免架构 | 243 |
| 第 11 章 大促保障体系 | 246 |
| 11.1 大促保障概述 | 246 |
| 11.1.1 大促保障简介 | 246 |
| 11.1.2 大促保障整体流程 | 247 |
| 11.2 大促保障体系详解 | 249 |



| | | |
|--------|--------------------|-----|
| 11.2.1 | 容量保障体系 | 249 |
| 11.2.2 | 风险保障体系 | 253 |
| 11.2.3 | 组织保障 | 255 |
| 11.2.4 | 运维保障 | 255 |
| 11.2.5 | 中间件保障 | 256 |
| 11.3 | 大促容量峰值保障策略 | 257 |
| 11.4 | 大促风险保障策略 | 259 |
| 11.4.1 | 风险保障概述 | 259 |
| 11.4.2 | 风险保障常见风险 | 259 |
| 11.4.3 | 风险识别和风险分类 | 260 |
| 11.4.4 | 风险保障策略 | 263 |
| 11.4.5 | 分组隔离策略 | 265 |
| 11.4.6 | 业务降级策略 | 265 |
| 11.4.7 | 监控发现策略 | 265 |
| 11.5 | 大促资金安全保障策略 | 265 |
| 11.5.1 | 常见的资金安全防护策略 | 265 |
| 11.5.2 | 大促资金安全防护 | 268 |
| 11.6 | 大促经验沉淀 | 268 |
| 11.7 | 大促保障实战分析 | 269 |
| 11.7.1 | 机房网络瓶颈问题分析 | 269 |
| 11.7.2 | 集群个体异常造成的容量问题分析 | 275 |
| 11.7.3 | 诡异的网络瓶颈 | 278 |
| 11.7.4 | 多机房压力测试流量不均问题分析 | 283 |
| 11.7.5 | Tengine 限流案例 | 291 |
| 11.8 | 总结 | 292 |
| 第 12 章 | 数据分析驱动性能优化 | 293 |
| 12.1 | WebP 性能优化案例背景 | 293 |
| 12.1.1 | WebP 格式开始兴起 | 294 |
| 12.1.2 | WebP 改造使 L-D 转化率下降 | 295 |
| 12.2 | 性能优化中的数据分析原理与方法 | 296 |
| 12.2.1 | 数据分析简介 | 296 |



| | | |
|--------|----------------------------------|-----|
| 12.2.2 | 数据分析之杜邦分析 | 297 |
| 12.2.3 | 数据分析之多维分析 | 299 |
| 12.3 | 通过数据分析来诊断 WebP 的性能问题 | 303 |
| 12.3.1 | 指标定义 | 303 |
| 12.3.2 | 基于指标树自动诊断 WebP 的性能问题 | 305 |
| 12.4 | 案例：通过数据分析进行 OLAP 分析和 RT 优化 | 308 |
| 12.4.1 | 在线分析系统响应指标基线的定义 | 308 |
| 12.4.2 | 性能问题诊断 | 309 |
| 12.4.3 | 数据的获取及觉察 | 311 |
| 12.4.4 | 方案的推导 | 313 |
| 12.4.5 | 小结 | 315 |
| 12.5 | 通过函数抽象进行性能优化 | 316 |
| 12.5.1 | 优化过程简介 | 316 |
| 12.5.2 | 函数抽象 | 317 |
| 12.5.3 | 统计分析 | 319 |
| 12.5.4 | 小结 | 321 |



1

第 1 章

基于用户体验的性能优化要素

1.1 页面用户体验的要素介绍

说到用户体验，它给人的第一印象总是：

- 抽象，带有强烈的主观意识。
- 难以量化。

想象这样一个场景，几个人在会议室里讨论一个产品功能的雏形，没过多久，会议室就充斥着各种你来我往：

- 我觉得这样会更好。
- 我认为用户喜欢这样或者那样。
-



到目前为止，用户体验已经渗透到很多领域，囊括了从硬件到软件的诸多极致体验设计。而最为大家熟知的，就是“苹果”手机了，从视觉、交互等细节上的设计，都能感受到“苹果”对用户体验的重视，并且好的用户体验能够作为一个非常重要的元素为产品加分，也切切实实带来了好的口碑。

在竞争激烈的电商领域，用户体验优化也能作为一个重要的要素，帮助我们脱颖而出。而网站性能的好坏，会直接影响用户的第一感觉，应该没有人能忍受一个经常卡顿的系统或者产品，即使它是“苹果”手机。

以现实举例，我们假设各个电商平台是一家家 SuperMarket 或者大商场，并且它们都拥有以下几个优秀 Market 的要素：

- 商品丰富、质量有保证等。
- 信誉好、品牌影响力大等。
- 价廉物美等其他因素。

假设 A-Market 是这样的：

- 拥挤不堪。
- 导购服务不到位。
- 结账让充满挫败感。

这些都令人生畏，对吧，即使它可能拥有更丰富的商品。

B-Market 是这样的：

- 井然有序。
- 导购服务快速到位，能帮助你更快地找到想要的商品。
- 商品详细介绍专业有效。
- 结账过程快速有效。

显然，消费者更倾向于到 B-Market 购物。

接下来回归我们的主战场，作为一家服务全球的跨境电商网站，它给予用户的服务体验与线下 Market 本质是相似的，并且会更加庞大。不像普通的区域化服务的本地 Market，我们要服务更多不同国家的用户，随之而来的是，不同国家复杂的网络环境带来的各种网站性能问题，我们需要在考虑投入产出比的情况下，尽可能让不同国家的用户在网站上都能有“快”的用户体验。



那么什么是“快”的用户体验呢？

我们不可能让所有东西都快起来，这样会让我们面临打破系统架构、破坏可维护性、甚至与商业规则冲突的风险，从而让我们举步维艰。所以我们需要找到关键的点，有针对性地做网站的性能优化，使效果最大化。

从用户打开浏览器、浏览网页的感知过程出发，我们分解出了下面几个要素来衡量网站性能方面的用户体验，并尝试通过不同监控系统进行系统监控和指标量化：

- 白屏。
- 首屏。
- 页面整体加载。
- 页面可交互。
- 功能交互响应。

下面我们会详细介绍一些指标要素，并介绍如何量化它们。

1.2 白屏时间

1.2.1 白屏时间的重要性

简单理解，即用户打开浏览器输入网站的 URL 后，从屏幕空白到第一个画面出来的时间，这个时间的长短将直接决定网站页面给用户的第一印象。

图 1-1 是一张页面加载过程的视频截图，可以很形象地反映白屏的过程。

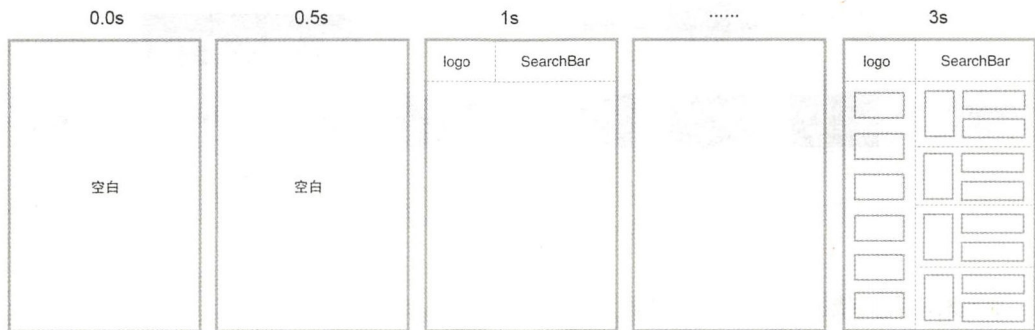


图 1-1



在很多商用或者开源的基调性能系统（比如 Webpagetest）中，将白屏时间命名为 StartRender，而在最新的 Chrome、IE 浏览器提供的 Performance Timing API 中也提供了类似的称为 FirstPaint 的指标。

StartRender 或 FirstPaint 代表浏览器开始渲染的时间，从用户的角度来看，就像图 1-1 中所表现出来的，即从用户在浏览器地址栏中输入 URL 并按 Enter 键开始，到在页面上看到第一个画面的时间间隔。

重要性：页面渲染的时间越短，用户等待的时间就越短，用户感知到的页面速度就越快，这样可以大大提高用户体验，减少新用户的跳出，提高留存率。反之，过长的等待时间，会让用户变得烦躁，更轻易跳出或者关闭这个网站。

1.2.2 白屏过程详解

白屏过程我们通过图 1-2 来说明。从中可以看到浏览器与服务器做了哪些交互，又在客户端经历了哪些过程。

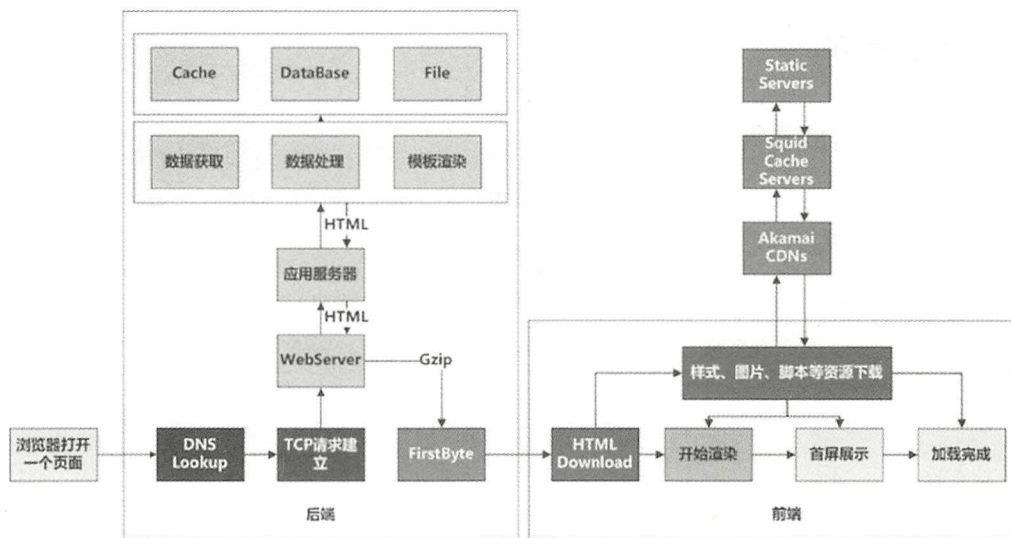


图 1-2

接下来，我们依次说明每个步骤。

1. DNS Lookup

简单地理解，DNS Lookup 即浏览器从 DNS 服务器中进行域名查询。浏览器解析域名拿到





对应的 IP 地址后，才能和服务器进行通信。通常浏览器在加载页面的过程中，会进行很多次 DNS Lookup 操作，其中包括页面本身的域名查询，以及浏览器在解析页面 HTML 代码过程中，需要加载 JS、CSS、Image 等资源产生的解析。

对大中型网站来说，通常在页面加载过程中会产生下列域名解析：

- www.example.com，页面 URL 本身。
- style.example.com，JS、CSS 等静态资源请求。
- img.example.com，静态图片请求。
- ajax.example.com，各种站内的 Ajax 请求。
- *.google.com，其他比如 Google 的站外请求。

域名解析的过程，可短至几十毫秒，也可能消耗几秒，所以解析过程的快慢直接影响页面整体加载的渲染速度，特别是对于每天访问量达千万级别的页面。

DNS 解析速度的优化策略有很多，通常我们会从以下几个方面思考细节的优化：

- DNS 缓存优化。
- DNS 预加载策略。
- 页面中资源的域名的合理分配。
- 稳定可靠的 DNS 服务器等。

后面会有专门的章节来详解 DNS Lookup 及其优化策略。

2. 建立 TCP 请求连接

浏览器和服务端 TCP 请求建立的过程，基于 TCP/IP，全称为 Transmission Control Protocol/Internet Protocol，该协议由网络层的 IP 和传输层的 TCP 组成。IP 作为每一台联网设备在 Internet 中的地址，是建立传输通道的基础。

浏览器针对各个域名进行 DNS Lookup 得到 IP 地址后，就开始建立 TCP 请求连接的过程。

TCP 通过三次握手建立连接，并提供可靠的数据传输服务。

我们可以通过下面两个要素理解 TCP 连接及传输的过程。

- 网络传输链路建立。
- 数据传输确认。

基于上面两个要素，我们简单梳理一下 TCP 优化的思路，后面会有专门的章节来详解 TCP





连接传输及其优化策略。

在真实的网络环境中，网络节点错综复杂，TCP 建立的网络传输链路并不一定如你所愿，不是每条链路都是高速公路，不是每条高速公路都能保持通畅。同时，数据在传输过程中不可避免会出现丢失或者被破坏的情况。

那么我们该如何应对呢？

- 基于不同的网络环境，优化数据包的大小，以减少数据因传输丢失或者被破坏产生的重传，从而提高传输效率。
- 网络传输链路的优化，对于大部分企业来说，是需要很大投入的。例如，尝试在部署在不同国家的后台服务系统之间，有针对性地建立高速的专属网络通道，或者通过购买第三方内容网络服务来加速浏览器和 Web 服务器之间的网络通道优化，可以想象投入是无比巨大的。如何整合各种优质资源，帮助我们达到链路优化的目标，会在 TCP 优化相关章节中讲述。

3. 服务端请求处理响应

在 TCP 连接成功建立后，Web 服务器接受请求，开始请求处理，而浏览器端则开始等待服务器的处理响应。

以一个动态的 HTML 请求举例，通常 Web 服务器在接收到此类请求后，会做出下面一系列处理。

(1) Web 服务器根据请求类型，将请求转发给已经注册该请求的应用服务器来处理。如果网站服务器的前端架构是通过 Apache+JBoss 来实现的，则 Apache 会将请求转发给后端的 JBoss 应用服务容器来处理。相反如果是一个未做过其他应用服务器注册请求类型的请求，则 Web 服务器会自己来处理。比如静态页面、图片等请求，Web 服务器会通过对文件系统进行 I/O，获取内容，然后跳过后面的步骤，直接进行 Gzip 压缩后输出。

(2) 应用服务器接收请求，分配给对应的代码单元处理。

(3) 从缓存、数据库、文件系统等获取数据。在大型网站的后台，通常采用分布式的服务架构，各个业务数据的获取是以中间件的形式通过各个相互依赖的服务相互调用的过程。

(4) 基于业务进行的数据逻辑处理，可以简称为业务逻辑层。

(5) 基于模板进行数据格式化渲染。

(6) 应用服务器将格式化渲染的数据返回 Web 服务器。





(7) Web 服务器将最终要响应 (Response) 到客户端的内容, 经过 Gzip 压缩 (通常网站管理员会开启这个配置) 后, 返回客户端。

通过列举这些过程我们可以看到, 服务端的请求处理速度优化, 是一个非常复杂的专题, 它涵盖的范围非常广, 并且对于不同的网络平台, 也有不一样的优化策略。

对于中小型网站来说, 重心可能主要放在数据获取过程的优化。其中对缓存、数据库等处理的优化, 可为网站页面带来较大的响应速度提升。

而对于有着亿级流量的大型网络平台, 每一个过程的优化都可能带来质的提升。比如被很多人忽略的模板渲染过程, 通过优化模板渲染逻辑, 可使模板渲染速度提高一倍; 或者通过调整 Gzip 算法及响应内容的代码结构, 来提高 Gzip 压缩率、减小压缩包大小等。

关于服务端的优化策略, 后面会有专门的章节来讲述。

服务端完成请求处理, 开始响应客户端内容 (在 Web 页面中, 客户端的角色主要由浏览器来扮演), 即开始了在前端由浏览器显示页面的过程。

4. 客户端下载、解析、渲染显示页面

服务器返回 HTTP Response 后, 浏览器陆续开始接收数据, 进行 HTML 下载、解析、渲染显示等过程。

具体步骤如下。

- (1) 如果是 Gzip 包, 则先解压为 HTML。
- (2) 解析 HTML 的头部代码, 下载头部代码中引用的样式资源文件或者脚本资源文件。
- (3) 解析 HTML 代码和样式文件代码, 这个过程会构造出两个树结构, 即与 HTML 相关的 DOM 树, 以及与 CSS 相关的 CSSOM 树。
- (4) 通过遍历 DOM 树和 CSSOM 树, 浏览器依次计算每个节点的大小、坐标、颜色等样式, 构造出渲染树。
- (5) 根据渲染树完成绘制的过程。

上面的渲染过程, 我们可以使用 Chrome 的控制台开发工具中的 TimeLine 进行观察, 如图 1-3 所示。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

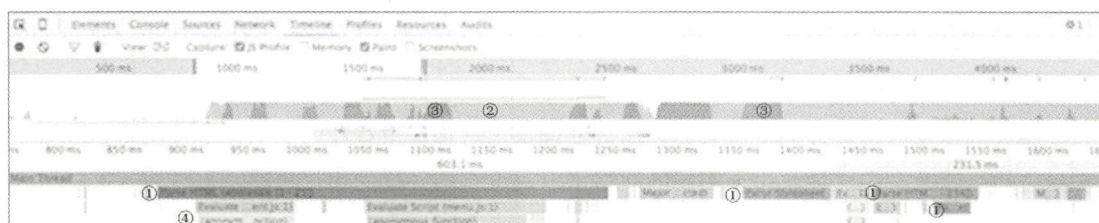


图 1-3

图 1-3 中①代表 ParseHTML，②代表 Scripting，③代表 RecalculateStyle 和 Layout 过程，④代表 Paint 过程。

基于上面的分析，我们可以知道理想的页面显示过程如图 1-4 所示。

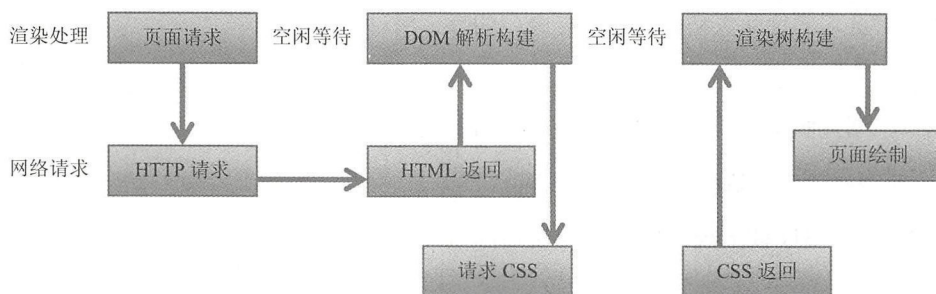


图 1-4

浏览器下载 HTML 后，首先解析头部代码，进行样式表下载，然后继续向下解析 HTML 代码，构造 DOM 树，同时进行样式下载。DOM 树构建完成后，立即开始构造 CSSOM 树。如果样式表的下载速度足够快，DOM 树和 CSSOM 树就进入一个并行的过程，当两棵树准备完毕，即可开始构造渲染树，最后进行绘制。

而在真实页面解析过程中，浏览器通常会因为各种因素被阻断。

(1) HTML 代码中的 JavaScript 代码（简称 JS 代码）会阻断 DOM 树的构造，因为浏览器认为这段 JS 代码可能会修改 DOM 结构，所以必须等待 JS 代码执行完毕，再恢复 DOM 树的构造过程。这是由浏览器的安全解析策略决定的，目前并没有指定某个 JS 代码不涉及 DOM 的属性。

(2) 浏览器必须等待样式表加载完成，才能开始构建 CSSOM 树。

(3) 还有一种特殊情况，浏览器在解析 HTML 时遇到 JS 代码，而此时 CSSOM 树还未构建完成，则浏览器会暂停脚本的执行（浏览器同时也会暂停继续向下解析 HTML 代码，从而导





致 DOM 树的构建过程被暂停阻塞），直到 CSS 样式文件下载完成，并完成 CSSOM 树的构建，才会重新恢复原来的解析。这也是由浏览器的安全解析策略决定的。

通过对上面各因素的分析，我们可以发现，HTML 中的内联 JS 代码执行的危害之处，不在于它阻断了 DOM 树的构建过程，除非是一段特别恶劣的 JS 代码运行非常慢。通常内联的 JS 代码运行大概消耗几十毫秒，也就是暂停构建几毫秒到几十毫秒。它最大的危害在于上面第三种情况提到的，即 DOM 树构建被阻塞的时间不是只有 JS 代码运行的时间，而是会加上样式资源文件下载和 CSSOM 树的构建时间，这时浏览器所进行的串行解析过程，与我们在前面期望的 DOM 树和 CSSOM 树^①的并行解析过程相差甚远。我们用图 1-5 来表示这个过程。

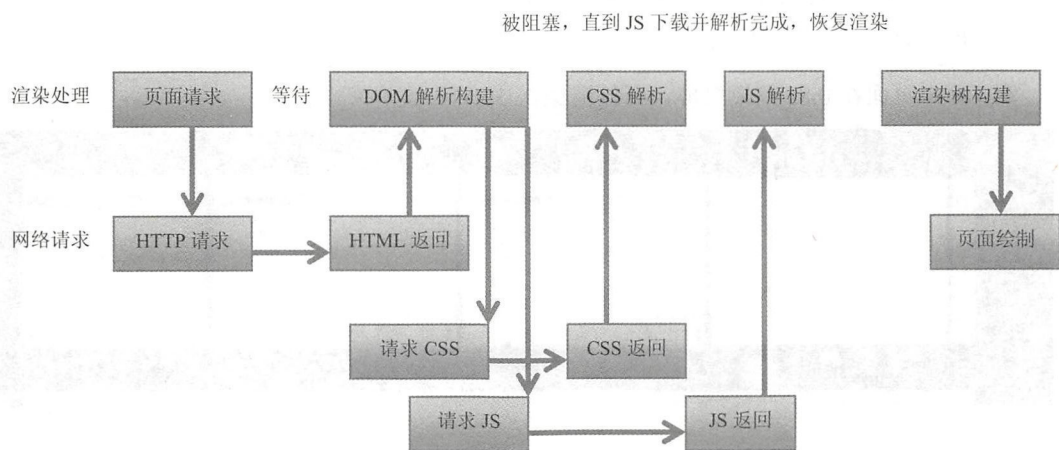


图 1-5

简单地对比图 1-4 和图 1-5，每一个步骤的时间段可能是一样的，但意外的阻塞等待导致总体消耗时间大大加长了。

所以，关于浏览器下载、解析、渲染显示页面的优化策略，根据渲染步骤，大概可以从以下几个方面着手：

- 优化 HTML 代码和结构，缩短 HTML 下载时间，加快 HTML 解析速度。
- 优化 CSS 文件和结构，缩短 CSS 文件下载时间和解析时间。

^① DOM 树即我们熟知的 Document Object Model 节点树，而 CSSOM 树是根据样式文件内容结构生成的。基于 CSSOM 树，可关联各个 DOM 节点的样式定义。具体可参考 <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=zh-cn> 中的解释。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

- 合理放置 JS 代码，避免前面第三种情况的出现，这也是最重要的。

1.3 首屏时间

1.3.1 首屏时间的定义

首屏，从字面上可以理解为“第一个屏幕”，而首屏时间即代表页面在“第一个屏幕”中的内容完全展示出来的时间。

图 1-6 和图 1-7 这两张页面加载过程的视频截图，可以很形象地反映页面从白屏到开始渲染（StartRender），再到首屏内容完全展示的过程。

图 1-6 中页面在 0.8s 结束白屏，开始渲染。

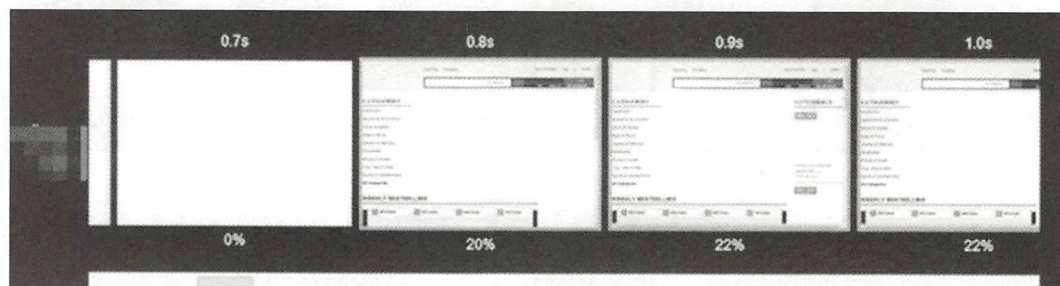


图 1-6

图 1-7 显示页面在 3.6s 左右完成首屏内容的展示。



图 1-7

受限于不同设备的分辨率的差异，不同用户看到的首屏也是有区别的，通常都由网站的负责人基于各自用户的访问设备情况，来定义一个普适性的标准尺寸，方便后期进行指标量化。

与白屏不同的是，我们并不能从浏览器的系统 API 中获取首屏时间。在真实网络环境中，





我们通过模拟的方式来获取首屏指标；而在实验室环境中，我们则可以通过一些性能测试工具来获取。这个我们在后面的指标量化里会做详细说明。

截至本书编写时，在 Chrome 60 Beta 的最新特性中包含了关于 Paint Timing API 的描述。

尽管没有公认的标准能够在所有情况下完美地反映页面的加载时间，First Paint 和 First Contentful Paint 仍然为衡量页面加载期间关键的用户参与环节提供了极具价值的。为了让开发者更好地洞察网站的加载性能，全新的 Paint Timing API 公开了捕获 First Paint 和 First Contentful Paint 的指标。

图 1-8 摘自 Google I/O 2017 上发布的《网络性能：挖掘最影响用户体验的指标》（*Web Performance: Leveraging the Performance Metrics that Most Affect User Experience*）。基于官方的描述，我们可以借助新增的指标来完善目前的量化标准定义。

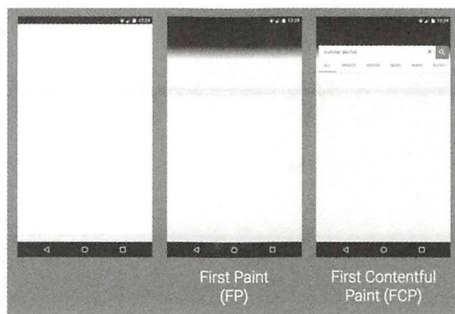


图 1-8

1.3.2 首屏时间的重要性

前面已经提到渲染的重要性，过长的白屏等待时间，会让用户变得烦躁，更轻易跳出或者关闭这个网站。通过观察图 1-8，我们很容易发现渲染时间指标的缺陷。

在该图中可以看到，在 0.8s 的时间点上，浏览器开始渲染首个画面（这是一个经过优化后表现不错的渲染时间），但用户看到的只是页面的整体框架，一些重要的内容并没有展现出来。

而首屏时间在页面整体加载的过程中，在开始渲染（StartRender）和加载完成（OnLoad）的中间，可以针对渲染时间指标的缺陷做出很好的补充，能够帮助我们综合衡量页面的性能情况，更真实地反馈页面带给用户的性能体验。

综上所述，我们可以从两个方面来理解首屏时间的重要性。





1. 完善指标量化系统

为什么它能起到完善的作用？我们举个例子，来看一下现实中的情况。

浏览器从 Web 服务器接收返回的 Response 后，开始按 HTML 代码结构从上至下解析并渲染。在未定义首屏指标之前，通常我们认为大部分页面的加载渲染过程会如我们预期的一样，从浏览器的窗口开始往下依次渲染。

假设页面加载渲染完成后，大概会有 3 个可视屏幕的高度，通常我们预期页面会如图 1-9 所示完成加载渲染过程。

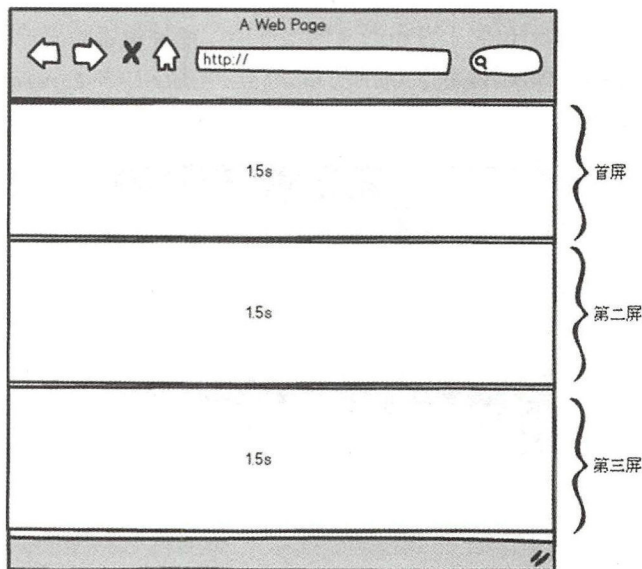


图 1-9

但通过 WebPagetest 对我们的页面进行性能测试后，得到的 SpeedIndex^①值并不如我们预期的一样快。我们可以看一下通过测试得到的首页加载过程的视频截图。

页面在 0.8s 结束白屏，开始渲染（StartRender），如图 1-10 所示。

① SpeedIndex 由 WebPagetest 提供，通过视频录制、画面到帧的分析来模拟检测首屏的渲染，可以反映在实验室中配置的各种不同终端及网络下，页面首屏内容展示完成的速度快慢。关于 SpeedIndex 的更多信息，可以查阅相关网站。

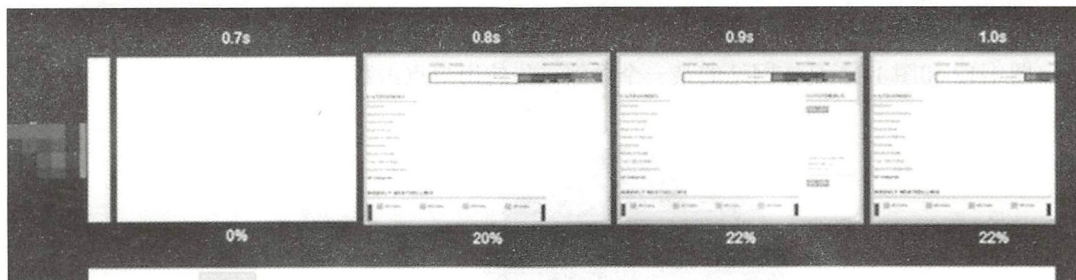


图 1-10

然后在 4.2s 左右完成首屏内容的展示，如图 1-11 所示。

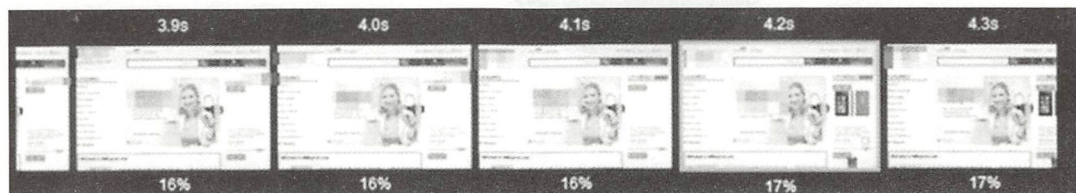


图 1-11

最后在 5s 左右完成了页面加载。

实际的测试结果表明，页面并没有如预期那样在 1.5s 左右完成首屏内容的展示。

我们在这里简单分析一下页面的性能。

在未加入首屏指标之前，我们使用 `StartRender` 和 `OnLoad` 来衡量这个页面的性能：在 1s 内开始渲染，最终花了 5s 左右的时间加载完成整个页面。从整体上看，是一个尚可接受的性能数据。

而在加入首屏指标之后，这个页面的性能描述是这样的：在 1s 内开始渲染，花了 5s 左右完成加载，用户整整等待了 4.2s，才看到首屏的内容。

很明显，页面存在一些隐含的因素，导致了如上描述的性能问题。

那么我们为什么要加入首屏指标，让原本性能尚可的页面一下子变成一个性能糟糕的页面呢？

下面我们看一些数据。



2. 更加真实地反馈用户体验

图 1-12 和图 1-13 是在网站的某一个页面采集的用户点击数据的分布图。

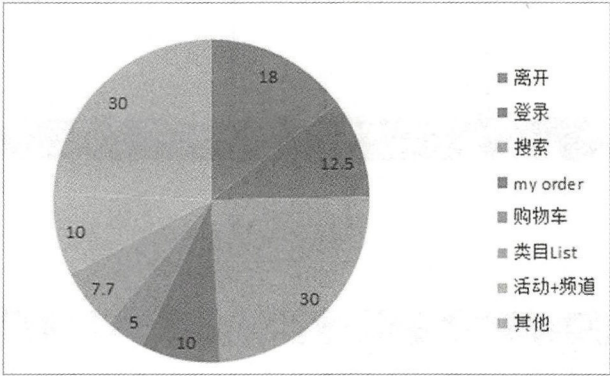


图 1-12

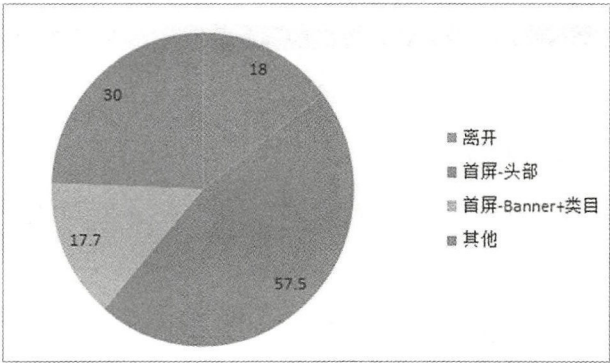


图 1-13

通过统计发现，70%以上的用户点击路径集中在首屏区域。

基于此数据，我们认为，至少在我们的电商网站上，页面首屏内容的展示速度会直接影响用户的性能体验。

如果在你的网站上，用户的浏览、操作也相对集中在首屏区域，那么就应该制定方案，有针对性地去首屏展示速度优化。

其实首屏的优化并不复杂，往往问题都出现在一些开发者容易忽视的细节上。比如不合理的 HTML 代码结构、页面资源下载序列等。最关键的是，要能够找到衡量各自页面首屏内容展示速度的办法，通过监控发现潜在的问题，然后做出调整。



1.4 页面整体加载完成

页面加载完成又称为 PageLoad，顾名思义，即页面相关资源（CSS 样式文件、JS 脚本文件、图片等）全部加载完成的时间。当这个时间点被触发时，意味着当前页面对于用户来说是可以进行安全交互的。

PageLoad 作为一个传统通用的性能指标，在页面性能衡量体系里已经被使用很久了。前面提到的 StartRender 和首屏性能指标，则是在 Web 不断发展、页面内容形式越来越多、页面功能交互越来越重的背景下，对 PageLoad 的一个非常重要的补充。

在描述 PageLoad 的时候，我们都希望知道多少时间的 PageLoad 才算快。

通常我们熟知的有“8s 规则”，虽然它还不能作为行业规则，但也能够作为一个重要的参考。这是在 20 世纪 90 年代进行的一项可用性实验研究得出的结论是，如果用户等待加载完成需要的时间超过 8s，用户会放弃对页面的进一步浏览，离开网站，从而导致用户流失。

随后在 2006 年，一个来自 Akamai 的新研究，发表了新的观点：通常 4s 左右的平均加载时间，是用户可能会等待页面加载的最长时间。

诚然，他们都通过实验或者调研，得出了各自对合理的页面加载时间的定义。但在真实的网络环境中，不同的网络环境（WiFi 或者 Modem）、不同网站的不同页面内容（偏展示或者偏交互）、不同页面的侧重点、不同的用户群体特征等，都会影响一个用户在浏览网页时对于 PageLoad 的忍耐度。

如果没有条件针对网站的性能指标和商业转化指标做相关性监控，找到网站的合理性能阈值，可以参考“8s 规则”或者“4s 规则”定义自己网站的性能基线，然后以此来衡量网站页面加载的快慢。



2

第 2 章

前端性能优化实战

通过前面的章节，想必大家已经熟悉了页面性能的几大要素，以及如何利用工具或者自建系统进行页面的性能分析。本章为大家介绍两个典型的优化实例。

2.1 延迟渲染

通常为了加快页面渲染的速度，基础的解决思路是，通过去除页面上除首屏以外的对于用户不可见的信息区块，让页面的 DOM 节点数更少、DOM 树结构更简单，从而达到加快页面下载和渲染速度的目的。

然后使用懒加载异步化请求、BigPipe 等方案，动态加载这些不可见的信息区块。

考虑 BigPipe 方案的实施成本，大部分情况下，笔者会选择使用懒加载异步化请求的方案对页面进行优化。



2.1.1 挑战和困难

在实际的页面优化过程中，并不是一切都如预期那样顺利。笔者曾遇到一个问题，即页面的主体部分不能再使用异步化请求的方式分割加载页面信息。

这个页面有什么特殊之处，为何它被拒绝使用异步化请求进行懒加载呢？因为它是网站重要的主流程页面，即承担搜索功能的搜索结果列表（List）页面。在很多电商网站中都可以发现，搜索页面的结构非常相似，通常会包含两大主体搜索结果内容，如图 2-1 所示。



图 2-1

从如图 2-1 所示的页面结构可以发现，页面主体是典型的左右两栏布局。其中左栏一般为关联产品搜索结果的类目数据，右栏为搜索结果数据。所以对于页面的首屏来说它包含了两部分，通常这两部分数据都来自于两次独立的数据查询的结果。

如果使用异步化请求分割右栏的搜索结果列表，当单页请求结果数为 50 时，首屏展示区域为 3 条结果。抛开这种请求方式对于服务端搜索引擎分页处理系统的挑战不说，一次搜索结果分两次请求获取，这对于服务端的资源也是很大的浪费。我们都知道，在服务端数据查询请求的优化中，强调的是对于查询请求资源次数的优化，而不是查询结果的大小。在这样的请求中，查询结果一次返回 3 条数据还是 50 条数据，所消耗的时间几乎是一样的。

表面上页面通过这样的分割，可以达到减少 DOM 树节点数、简化 DOM 树结构的目的，但考虑到服务端所付出的代价，这无疑得不偿失。

所以在这样的情况下，我们必须寻找其他方式，来简化这个庞大的搜索结果列表页面。

2.1.2 解决方案

既然服务端的解决方案有诸多限制，那么将焦点转移到浏览器端。笔者希望在尽量不改动服务端在 Web 应用层的 HTML Template 输出逻辑的前提下，在浏览器端找到能够加快页面渲



染速度的办法。

注：之所以要求尽量不改动 Web 应用层的 HTML Template 输出逻辑，是为了保障搜索页面模板代码的可维护性。

首先我们大概回顾下浏览器端的页面渲染过程，如图 2-2 所示。

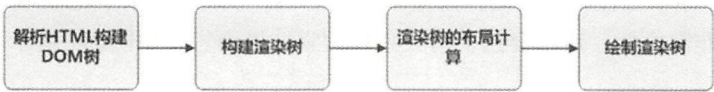


图 2-2

其中渲染树即之前提到过的 CSSOM 树的构建。基于这个渲染过程，我们通过各种测试来尝试找出在搜索页面中真正的渲染瓶颈在何处。

为了方便大家理解测试的过程和结果，我们假设一个前提，完整的搜索结果页面包含 36 个搜索结果。其中首屏展示的部分包含 3 个结果，首屏以下不可见部分则为剩下的 33 个结果。

图 2-3 是笔者汇总的测试报告。

| 原因猜测 | 测试方法 | 结果 |
|-----------|---------------------------|-----------------------------|
| 绘制Render树 | 逐步隐藏List页面的区块 | 能使StartRender变快，但效果不明显 |
| 构建Render树 | 逐步减少List页面的CSS文件的代码量，直到没有 | 每一次逐步减少，使List StartRender变快 |
| 构建Render树 | 逐步减少List页面的DOM节点数量 | 每一次逐步减少，使List StartRender变快 |

图 2-3

下面来详解每个测试结果。

1) 将矛头指向“绘制 Render 树”

在测试过程中，逐步隐藏搜索结果，白屏等待的时间有所减少，但效果不明显。

2) 测试“Render 树构建”

在测试过程中，逐步删减页面 CSS 文件的样式代码直到没有。笔者观察到，每一次减少都能使 StartRender 时间缩短，直到最终页面无任何样式代码时，StartRender 时间与原先相比大幅缩短。通过测试发现，基于 DOM 树和样式构建 Render 数据的过程也非常耗时，但是页面不可



能做空版页面，CSS 文件精简的程度有限，于是紧接着做第三个测试。

3) 观察 DOM 树

在测试过程中，逐步减少搜索结果数量，直到剩下首屏展示的三个结果，得到的测试结果和上一个测试结果相似，而且 `StartRender` 的优化程度更明显。基于渲染过程，笔者对于结果的分析是，DOM 树节点数量的减少不仅能够加快 DOM 树的构建过程，而且也能加快 Render 树的构建过程。

通过上面不同维度的测试，笔者发现在整个渲染过程中，主要的消耗还是集中在 DOM 树和 Render 树的构建上。其中 DOM 树的复杂程度对于整个渲染过程的复杂程度起到关键作用，浏览器解析文档构建 DOM 树，然后遍历 DOM 树，计算每个 DOM 树节点的样式，生成 Render 树。我们可以用一个简单但并非完全准确的公式来表现这个过程的复杂度。

在不考虑 CSS Selector 深度的情况下：

$$\text{DOM 树节点数} \times \text{CSS Selector 数} = \text{遍历循环次数}$$

以我们的页面举例，大概有 2500 个 DOM 树节点和 2000 个 CSS Selector：

$$2500 \times 2000 = 5\,000\,000$$

最多可能会有 500 万次的遍历循环。

通过上面的测试分析，我们明白了浏览器在 DOM 树上构建 Render 树并不是非常轻松的任务。如果你希望开发一个高性能的 HTML 页面，在编写 HTML 和样式的时候，都应该加倍小心，尽量精简结构，并且要避免通过滥用样式选择器得到高效的 CSS Selector。

回到我们的目标页面，在不影响业务功能展现的前提下，我们的页面已经采集了很多基础方案进行了优化。例如 HTML 结构精简、样式文件优化、CDN Cache、DNS Prefetch 等，但还是未能达到“1s 内开始 `StartRender`”的目标值。

基于上面的测试结果分析，可以判断出目标页面的渲染性能瓶颈在于，页面的 DOM 树节点数过多，导致在 DOM 树和 Render 树构建上所消耗的时间过多，从而使页面渲染性能不够高。

所以现在方案也非常明确了，就是要减小页面首次渲染时的 DOM 树节点数，并且在不修改服务端输出逻辑的前提下进行。

然后细化分解一下，希望提供的方案能够将首屏以下不可见的 33 个结果，在浏览器首次渲染的时候，在页面的 DOM 树中消失，达到加快首次渲染速度的目的。在其处于可见状态的时



候，再将其恢复到 DOM 树中。

在考虑尽量不修改服务端逻辑的前提下（保障可维护性），笔者最终选择使用 `TextArea` 来存放首屏以下的 33 个搜索结果对应的 HTML 代码。存放在 `TextArea` 中的 HTML 代码，浏览器会解析识别为 `TextArea` 内容，而不会被当作 DOM 节点进行解析。所以通过这个办法，页面首次渲染的 DOM 树包含的节点数大幅减少，从而大幅提高首次渲染速度。

我们额外要做的事情是：在服务端，将一些特殊字符（比如 `&`）进行 `HTMLEncode` 转义。而在浏览器端，则需要在首屏以下区域处于可见状态时，将 `TextArea` 中的 HTML 代码取出，将其恢复到 DOM 树中进行渲染。

在这里，笔者将这个方案称为“延迟渲染”。在实际运用过程中，根据不同情况，也可以选择使用 `Script Tag` 来存放 HTML 代码。

我们看一下使用“延迟渲染”给页面带来的变化，如图 2-4 所示。

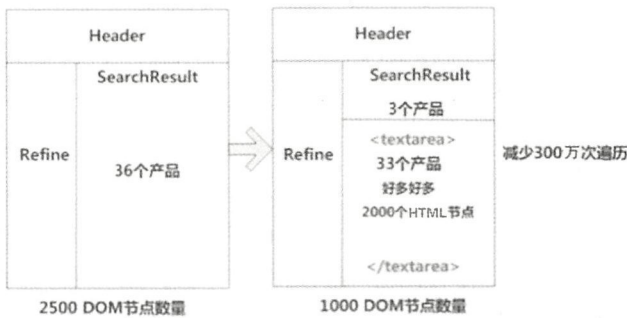


图 2-4

最终通过这个方案，我们解决了搜索页面的渲染性能瓶颈问题，将页面的 `StartRender` 稳定在 1s 内。

2.2 SEO Ajax

如果你的网站也是面向海外用户的，你肯定会和 Google 有很多的交集，特别是在 Google SEO 的流量导流优化方面。

通常对于一个处在发展早期的网站来说，尤其是跨境服务网站，都会非常依赖从 Google



SEO 渠道获取的所谓免费流量导流。所以很多大中型网站建立独立的页面 SEO 优化组织，长期持续地优化网站页面，来保障页面面向 Google 爬虫时是非常友好的，从而帮助爬虫更好地理解所抓取页面的内容。通过这种持续优化跟踪，来持续增加该网站在 Google 搜索结果里的排名，从而增加 Google SEO 渠道流量。

但随着业务的发展，页面的建设开发维护很有可能还会持续向 SEO 妥协，虽然保持了对爬虫的友好，但相对真实用户的性能体验却在下降。遇到这样的问题，该如何选择，或者有其他两全其美的方案吗？

2.2.1 挑战和困难

针对 SEO 页面的性能优化，会遇到什么问题呢？

- 针对爬虫爬取收录的页面，页面内容必须同步展示。
- HTML 代码和实际展现的内容要保证一定程度的一致性，否则有被认作作弊页面的风险。

在同步展示的要求上面，服务端要在一次请求过程中，把整个页面所有的内容都计算完毕，这就导致了页面在服务端的响应会消耗很多时间。同时一致性的要求，使得在浏览器端方面，我们也不能轻举妄动，极大地限制了各种优化方案的使用。

在必须保障 SEO 渠道流量不受影响的前提下，针对 SEO 页面的性能优化一度处于举步维艰的境地。

2.2.2 解决方案

Google 官方提供了一个称为 Ajax crawlable scheme 的 Ajax 爬行方案，是 Google 为了帮助爬虫抓取网络上越来越多的富应用页面产生的内容。

1. Ajax crawlable scheme 细节概括

简单地说，整个解决方案是这样工作的。爬虫在页面中发现 pretty Ajax URL（基于约定，URL 中包括一个“#!”这样的锚点）。接着爬虫会对这个 URL 做简单的修改，来表明意图：我是来抓取快照的。然后请求服务器。这个时候服务器返回一个包含全部内容的页面快照，由爬虫进行处理。而在搜索引擎结果中，只会展示原先的 pretty Ajax URL。

方案如图 2-5 所示。

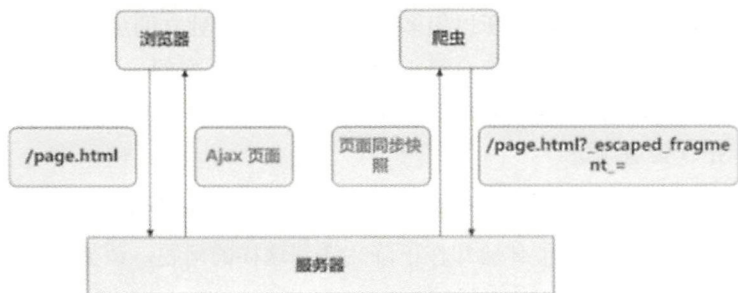


图 2-5

在实际方案运用中，会有两种不同的实现步骤：

- 通过改造 URL，向爬虫表明页面实现了 Ajax crawlable scheme。
- 通过页面 meta 标签，向爬虫表明页面实现了 Ajax crawlable scheme。

2. 改造 URL

改造 URL 的方案，比较适合原本就是富应用实现的页面，或者希望改造页面使其对爬虫更友好的情况。富应用页面中的很多内容都是通过 Ajax 异步请求的方式来动态生成的。

所以通过改造 URL，让爬虫在抓取原页面内容时，转去抓取页面同步快照。

如果富应用页面 URL 本身就已经使用了 Hash 来构造浏览器端的 queryString：

```
www.example.com/info.html#name=mp3
```

那么通过简单的改造，爬虫就可以知道该页面实现了 Ajax crawlable scheme：

```
www.example.com/info.html#!name=mp3
```

而当爬虫在抓取页面的过程中遇到上面的 URL 时，则会将其转换成：

```
www.example.com/info.html?_escaped_fragment_=name=mp3
```

在服务端要做的就是，针对转换过的 URL 请求，返回包含全部内容的页面快照。

这里所谓的“页面快照”可以理解为，应该包含页面中 JavaScript 代码执行完以后出现在页面中的所有内容。

3. 页面 meta

对于没有 Hash 的页面，或者不想修改网站的 URL 结构，担心 SEO 在老的 URL 权重建设上归零的站点来说，笔者也提供了另一种方案。



例如，页面原本的 URL 是这样的：

```
www.example.com/info/mp3.html
```

在页面的 head 标签中，添加如下 meta 标签内容：

```
<meta name="fragment" content="!">
```

当爬虫在抓取页面过程中遇到上面的 URL 时，则同样会将其转换成：

```
www.example.com/info.mp3.html?_escaped_fragment_ =
```

同样，在服务端要做的就是针对转换过的 URL 请求，返回包含全部内容的页面快照。

如果 Google 提供的方案可行，它将会带来很大价值：

- 可以放心使用 Ajax 异步化方案，而不用担心影响 SEO。
- 可以基于此方案，进行更深入的前台和后台优化。
- 采用此方案后，针对终端用户的响应和爬虫的抓取响应，可以在后台架构上做分离。在架构分离的前提下，针对终端用户，提供实时信息的响应；针对爬虫，提供一定时间缓存版本的响应。从而提高对爬虫的响应速度，最终减小来自爬虫的抓取压力。

作为网站的管理员，不管所在网站来自 Google SEO 的导流是大还是小，都需要承担应用新方案所带来的风险。所以明白它所带来的风险，从而循序渐进地测试方案，直到全量铺开，才是正确的方法。

4. 风险

Google 官方博客表明，他们会相似地对待采用此类方案的 Ajax 页面，就像对待其他 Web 传统页面一样，只获取爬虫通过爬行方案提供的方式抓取到此类 Ajax 页面的完整内容，但不保证此类页面在搜索结果中的排序。一切取决于内容。

Google 没有明确表示网站使用该方案可能产生的影响，如果网站非常依赖 Google SEO 渠道流量的导流，此部分流量的丢失对网站会有很大影响。

所以笔者带着下面的疑问去尝试和实验，进而控制风险：

- 方案是否为 Google 实验性方案，能否投入生产环境中。
- 对于页面 SEO 引流效果的影响的未知。

最后笔者经过半年的实验，验证了方案的可行性。并且在实验过程中，方案的应用还增加了 SEO 渠道的流量。



3

第 3 章

网站性能分析

本章介绍如何借助站外优秀的性能测试工具，以及建立网站自己的真实用户性能监控系统，来测试和监控页面的关键性能指标，从而使我们能够对网站性能问题进行快速分析并做出优化。

3.1 快速了解网站性能

借助外部各种优秀的免费工具和网站，可以快速了解当前网站的性能好坏，可使用的性能测试工具非常多。

3.1.1 使用 YSlow 进行性能分析

看过 Steve Souders 所著的《高性能网站建设指南》这本经典性能优化书籍的人，应该都非常熟悉 YSlow 性能检测插件，它以 Firefox 插件的形式，几乎成为了所有前端开发人员的标配



工具。YSlow 遵守基于所有基础的性能最佳实践的规则，快速分析 HTML 文档代码组成，Steve Souders 在《高性能网站建设指南》一书中总结了 12 条基本规则：

- 尽量减少 HTTP 请求。
- 使用 CDN。
- 静态资源使用 Cache。
- 启用 Gzip 压缩。
- JavaScript 脚本尽量放在页面底部。
- CSS 样式表放在顶部。
- 避免 CSS 表达式。
- 减少内联 JavaScript 和 CSS 的使用，尽可能使用外部的 JavaScript 和 CSS 文件。
- 减少 DNS 查询。
- 精简 JavaScript。
- 避免重定向。
- 删除重复的脚本。

这些规则能够帮助开发者快速地发现页面性能问题。一个页面如果遵守了以上规则进行前端开发，那么抛开网络因素，至少可以保证它在页面加载阶段的性能不会产生问题。

但我们回过头回顾一下这些实践规则，基本上它将焦点都放在了页面的加载阶段，而在加载完成后的浏览器的解析、渲染阶段是空白的。

对于一个常规的相对简单的页面来说，以上规则的检查足够了，但是一旦遇到包含大量 JavaScript 文件及复杂布局的页面时，很难通过 YSlow 来分析浏览器在页面解析、渲染各个阶段的时间，从而导致优化无从入手。所以接下来，我们看一下 Google 开源的优化工具 PageSpeed。

3.1.2 使用 PageSpeed 进行性能分析

和上面提到的 YSlow 一样，Google 提供的优化工具 PageSpeed 除了有网站的入口，也能够以插件形式在 Firefox 和 Chrome 浏览器上运行。它的工作原理类似于 YSlow，也是基于一些基础的性能最佳实践规则来分析代码，然后提供一系列能够提高页面性能的优化方案建议。

PageSpeed 也有它的独到之处。它除了能够提供页面在请求加载层面的优化建议，还能提供页面在加载完成后的一系列解析渲染操作的各部分时间，比如：

- 页面包含的 JavaScript 代码的执行时间。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

- 合理高效的 CSS 样式代码的建议。
- 页面布局渲染的时间等。

使用 Chrome 的控制台开发工具中的 TimeLine 可以看到类似图 3-1 的效果。

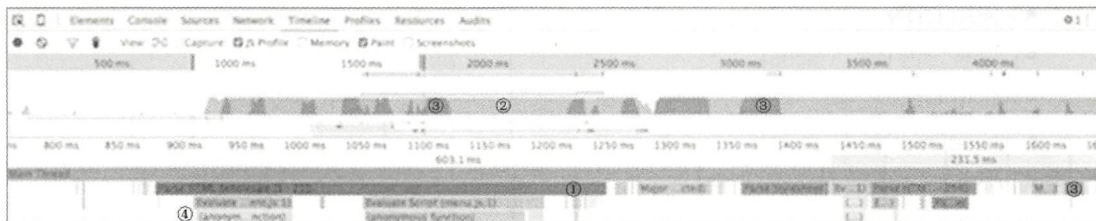


图 3-1

图中①代表 ParseHTML，②代表 Scripting，③代表 RecalculateStyle 和 Layout 过程，④代表 Paint 过程。

有了这些数据，我们就可以知道特定页面在浏览器解析渲染时，在哪里消耗了较多时间，然后对其进行优化。

然而，在现实世界里，用户在浏览页面时的性能体验，并不一定如我们所预期的那样（在本地开发完成并通过性能检查工具检查确认为具有良好性能表现的页面，并不代表用户在真实网络中访问会具有相同的表现）。

假设我们在本地开发页面，并利用上面提到的性能检查工具辅助我们写出具有良好性能实践的页面，然后发布更新交付给用户。但在真实的网络世界里，不同地域的用户、不一样的网络接入提供商、相差很多的网络速度等因素，都直接影响页面的加载速度。

特别是对于做跨境服务的电子商务网站来说，在本地开发出一个具有良好性能的页面是远远不够的。我们需要知道不同地域的用户访问网站的真实性能情况，比如美国和巴西的用户在访问同一个页面时的性能差别，我们购买的 CDN 服务在不同地域是否都起到正向的网络加速作用等。

当某个地区的用户向产品负责人反馈，他们打开网站很慢，而你只能无奈地在本地又重新打开 YSlow 或者 PageSpeed 扫描页面，然后得出结论：“我们这里打开还好，非常快。”

所以我们需要有一种工具，能够帮我们漂洋过海，去了解某个特定地区用户访问网站的真实性能情况。接下来看一下 WebPagetest，看看它能帮助我们做哪些事情。





3.1.3 使用 WebPagetest 进行性能分析

和 YSlow、PageSpeed 一样，WebPagetest 同样也使用和它们相似的一组最佳性能实践来分析网页。

WebPagetest 通过浏览器访问，基于输入的 Website URL，以及选择的国家城市、浏览器类型、网络带宽等信息，启动对应的远程服务器上的浏览器进行性能分析测试。

正如前面提到的，我们希望有一种工具，能够帮我们漂洋过海，去了解某个特定地区用户访问网站的真实性能情况。WebPagetest 就能够提供遍布世界各个角落的代表性城市的页面性能分析测试，基本覆盖亚洲、大洋洲、非洲、欧洲、北美和南美洲。

据官网介绍，他们提供的不同城市的性能分析测试，是通过真实的部署在对应城市的物理服务器来实现的，而不是通过其他类似代理或者虚拟机等方式模拟实现的。

除了常规的性能分析及优化建议，WebPagetest 还提供了页面加载过程的视频录制、关键节点截图、页面加载瀑布图等诸多重要信息。

如图 3-2 所示，在性能报表的概括里，有几个重要的性能指标：

- First Byte 表示首字节响应时间，该时间可以综合反映出当前连接的网络状况和服务器的响应处理速度。
- Start Render 表示浏览器开始渲染的时间。
- Speed Index 表示一种在 WebPagetest 中自定义的性能指标。
- Load Time 表示加载时间。

其中 Start Render、Speed Index、Load Time 分别代表了前面提到的几个重要的影响用户性能体验的性能指标：

- 白屏。
- 首屏。
- 页面整体加载。

使用 WebPagetest 测试页面时，通过对比页面在不同国家的城市中这几个指标的差别，可以大概知道这些国家的用户访问页面的真实体验是怎样的，用户等待空白页面花了多少时间，整个首屏显示完成又花了多少时间，最终用户等待了多少时间才可以正常浏览和使用页面功能。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

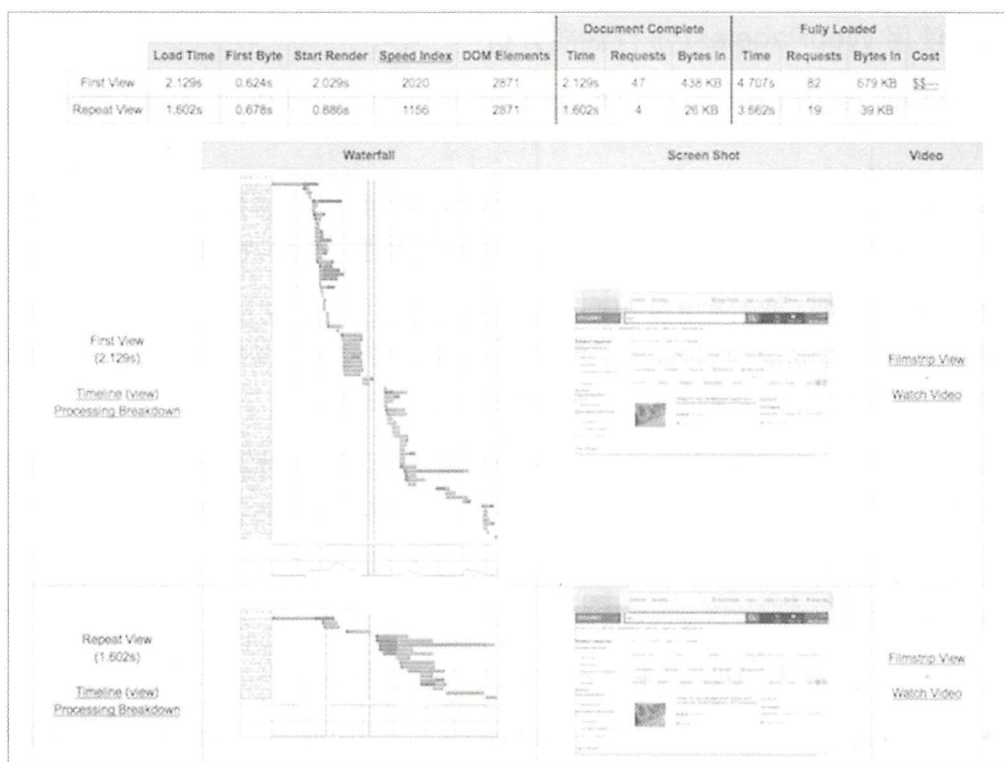


图 3-2

通过上面的介绍，从 WebPagetest 提供的这些功能来看，它的确是非常优秀的性能分析工具，但它的缺点也显而易见。

- WebPagetest 很难对登录态依赖 Cookie 的网页进行测试，基本上无法为测试页面设置正确的 Cookie。
- WebPagetest 上的性能测试服务是由人工触发的。每一次测试服务，都是在特定地区的特定机器上启动特定浏览器进行性能分析的，它的分析结果至多只能作为排查某个国家用户访问性能问题时的一个性能报告参考，并不能真正代表访问网站的真实用户的性能体验。

通过上面的介绍，我们发现 WebPagetest 虽然能提供强大的性能分析服务，但作为免费的测试服务，提供的测试样本数量有限，测试结果也不能真正关联真实用户。

而且如果网站已经过了初创时期，正处在稳步上升的阶段，用户的访问量逐步上升，达到了百万、千万级别，更不可能将网站日常的性能分析和监控，寄望于每日通过 WebPagetest 手





动测试得到的样本数量在个位数的性能报告。

所以，如果希望网站能够持续为用户提供最佳的性能体验，就需要建立网站自身的性能监控系统。

下一节笔者会给大家介绍一个基础的真实用户性能监控系统的设计，具有日常页面性能监控、预警等功能，并且能够在网站性能发生问题时提供足够的基础性能分析数据。

3.2 真实用户前端性能监控

建立一个基础的真实用户前端性能监控系统，大致包含以下 5 个系统模块的设计开发工作：

- 真实用户前端性能数据采集。
- 采集数据存储。
- 监控系统指标定义及加工计算。
- 数据分析、性能报表产出。
- 性能基线定义。

本节着重介绍“数据采集”和“监控系统指标定义及加工计算”。在解决了数据问题后，大家都可以根据实际情况产出自己想要的性能报表，用于分析监控。

3.2.1 真实用户前端性能数据采集

在分析如何采集真实用户性能数据前，先看一下我们想要得到的数据。基于前面章节介绍的几个关键性能指标，我们希望看到的数据除了基础的终端环境数据，都应该以采集到真实用户打开页面的网络下载消耗、白屏等待、页面加载完成等关键数据为目标来实现系统的数据采集模块。

假设系统设计的目标是希望能够得到下面的数据。

1. 终端环境数据

- 设备信息，用于区分 PC、Mobile、平板等不同设备。
- 操作系统。
- 浏览器。
- 地区/国家等。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

2. 基础性能指标数据

网络相关时间：

- 页面域名解析时间（DNS Lookup Time）。
- TCP 建立连接时间（TCP Connection Time）。
- 页面首字节时间，可理解为页面请求等待时间（Time To First Byte）。
- HTML 文档下载时间（HTML Download Time）。

浏览器端渲染相关时间：

- 页面开始渲染时间，即白屏等待时间（StartRender Time）。
- 文档对象模型准备时间（Dom Ready Time）。
- 页面加载完成时间（Page Load Time）。

3. 页面资源加载详细数据

页面加载包含的 IMG、JS、CSS 资源的时间消耗数据。

3.2.2 数据采集可行性分析

1. 终端环境数据采集分析

通常终端环境的数据通过浏览器的 DOM API 提供的 `navigator.userAgent` 属性都可以获取。这个属性声明了浏览器用于发送 HTTP 请求的用户代理头的值，它包含的信息如下。

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2661.102 Safari/537.36
```

可以通过解析日志请求中的 HTTP Request Header 的 IP 信息，在后期通过相关 IP 库拿到对应的地区或者国家信息。

2. 页面性能数据采集分析

如果网站用户大部分都在使用一些老旧的浏览器，比如 IE6、IE7、IE8 等，就不得不兼顾这些浏览器，只能通过在 HTML 文档中插入各个自定义的占位 JavaScript 脚本，来模拟获取页面加载过程中浏览器渲染的各个时间，并且获取不到页面加载过程中的网络端的时间消耗，示例代码如图 3-3 所示。

我们需要通过在文档中不同的位置侵入式地插入脚本来记录各个时间点，过多的侵入式脚本本身也会影响浏览器解析页面的速度。





```
<!DOCTYPE html>
<html>
  <head>
    <script>
      var pageStartTime = new Date().getTime();
    </script>
    <script src="your script file path"></script>
    <link rel="stylesheet" type="text/css" href="your style file path">
  </head>
  <body>
    <script>
      // 模拟StartRenderTime
      var pageStartRenderTime;
      var image = new Image();
      image.onload = function() {
        pageStartRenderTime = new Date().getTime();
      };
      image.src = 'https://[url]start-render.png';
    </script>
    <div>
      // 页面内容主体
      ....
    </div>
    <script>
      // Window Onload Time
      var pageLoadedTime;
      window.onload = function(){
        pageLoadedTime = new Date().getTime();
      }
    </script>
    <script>
      // 时间计算
      var startRenderTime = pageStartRenderTime - pageStartTime;
      var pageLoadedTime = pageLoadedTime - pageStartTime;
    </script>
  </body>
</html>
```

图 3-3

如果遇到这个问题，可以尝试创建一个项目来专门推动用户升级浏览器。对于用户来说，升级浏览器也意味着页面性能体验的大幅提升，想象一下，同样的页面在 IE6 和现在非常流行的 Chrome 浏览器下打开的速度和体验差别，就知道这样做能带来的收益有多大了。

所以在这里笔者着重为大家介绍，如何使用很多新版浏览器中已经被支持实现的 HTML5 Performance API 来获取更多真实的性能数据。

1) 基于 H5 Performance Timing API，收集页面的基础性能信息

Performance Timing API 主要包含 3 个部分。

- Navigation Timing: 主要提供页面加载过程的性能数据。
- Resource Timing: 主要提供页面所包含的脚本、样式表、图片等资源加载的性能数据。
- User Timing: 方便为在复杂脚本内部记录不同代码片段的执行时间提供 API 实现。

在这里笔者主要使用 Navigation Timing 和 Resource Timing 两个部分的 API，通过它们提供的属性数据，基本上可以获取页面从加载开始到结束涉及的网络加载及浏览器端渲染等所有时间消耗。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

(1) Navigation API。

通过访问 `window.performance.timing` 对象中的属性，可以得到当前页面的相关性能信息。笔者引用一张在 W3C 规范中定义的图来看一下 Timing 接口提供了哪些属性，如图 3-4 所示。

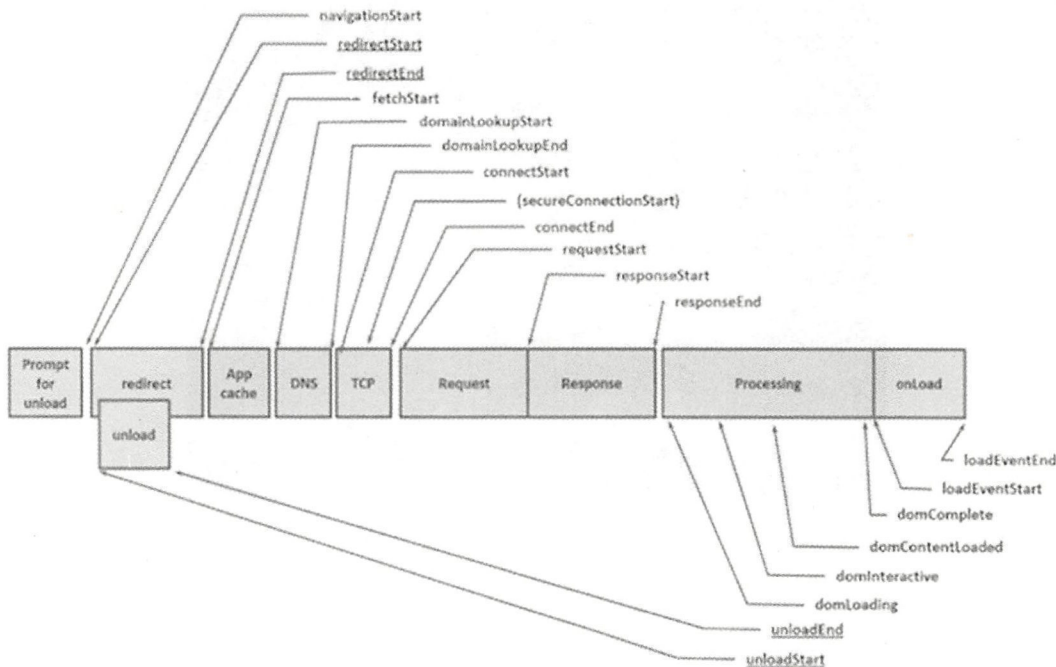


图 3-4

通过对照页面加载时序图，可以清晰地了解页面的不同加载阶段。比如计算域名 DNS 解析、TCP 请求等处理过程时间所需要的属性区间。

Timing 接口中提供的属性，都是 UTC 时间，即指定的时间距 GMT 时间 1970-01-01 午夜的毫秒数（在下面的介绍中，如果没有特殊说明，默认都是 UTC 时间）。一般我们都以 `NavigationStart` 或者 `FetchStart` 的值作为页面加载的开始时间，以 `ResponseEnd` 作为分隔点，在它之前的时间一般归属到网络相关的消耗，而在它之后的时间一般归属到终端渲染的消耗，最后以 `LoadEventEnd` 作为结束时间。

(2) Resource API。

通过访问 `window.performance.getEntriesByType("resource")` 返回的数组对象，可以遍历当前





页面所包含的资源加载相关的性能信息。

通过官方的例子发现,笔者获取某种特定资源的加载相关性能信息也非常方便,如图 3-5 所示。

```
<!doctype html>
<html>
<head>
</head>
<body onload="loadResources()">
<script>
function loadResources()
{
    var image1 = new Image();
    image1.onload = resourceTiming;
    image1.src = 'https://www.w3.org/Icons/w3c_main.png';
}

function resourceTiming()
{
    var resourceList = window.performance.getEntriesByType("resource");
    for (i = 0; i < resourceList.length; i++)
    {
        if (resourceList[i].initiatorType == "img")
        {
            alert(resourceList[i].responseEnd - resourceList[i].startTime);
        }
    }
}
</script>

</body>
</html>
```

图 3-5

2) 页面白屏时间的监控

很遗憾,在 Navigation API 规范定义中,并没有提供关于白屏(StartRender,浏览器开始渲染)的时间度量属性。

但幸运的是,目前在 IE9+、Chrome 浏览器中分别提供各自版本的 FirstPaint(即 StartRender)时间,可通过 JavaScript 访问来获取。

(1) IE9+。

在 IE9+中,页面开始渲染的时间由 window.performance.timing.msFirstPaint 来提供,即以毫秒为单位的 UTC 时间。

(2) Chrome。

在 Chrome 中,页面开始渲染的时间由 window.chrome.loadTimes().firstPaintTime 提供,但 Chrome 中返回的 FirstPaintTime 却是以特殊的秒、毫秒(seconds.milliseconds)^①为单位的 UTC

① 在前面的章节中笔者提到过,在目前最新的 Chrome 60 beta 版本中发布了 Paint Timing API,里面新增的两个指标 First Paint 和 First Contentful Paint 能够为当前量化体系提供更加完善的补充,方便开发者更好地洞察网站的加载性能。



时间。所以在使用它之前，需要做简单的处理，如图 3-6 所示。

```
> window.chrome.loadTimes().firstPaintTime
◁ 1469339689.118927
> window.chrome.loadTimes().firstPaintTime * 1000
◁ 1469339689118.927
```

图 3-6

3. 页面性能数据加工计算

从浏览器获取原始的性能数据后，开始对数据进行加工计算，来映射前面定义的和用户体验相关的各种体验指标。

1) StartTime 基准计算

StartTime 的定义非常重要，因为后面所有的指标都需要以此时间为基准进行计算。基于我们的经验，如果 NavigationStart 不为 0，则使用 NavigationStart 作为页面 StartTime 基数，如果为 0，则降级使用 FetchStart，如果 FetchStart 还是 0，则将考虑此数据作废。

2) 其他指标加工计算准则（见表 3-1）

表 3-1

| 目标指标 | 计算方式 | 定 义 |
|--|------------------------------------|------------------------|
| DNS解析时间 | domainLookupEnd-domainLookupStart | 从发起页面域名解析至解析完成 |
| TCP建立连接时间 | connectEnd-connectStart | 从发起TCP连接至三次握手完成 |
| 请求等待时间 | responseStart-requestStart | 从发起页面请求至服务器端返回第一个字节 |
| 文档下载时间 | responseEnd-responseStart | 从接收服务器发回的第一个字节至主页面下载完成 |
| 备注：以上都为各时间段耗时，以下都是total_time，以StartTime为基准 | | |
| 首字节时间 | responseStart-StartTime | |
| 页面DomReady | domContentLoadedEventEnd-StartTime | |
| 页面首屏时间 | FirstScreenTime-StartTime | |
| 页面加载完成时间 | loadEventEnd-StartTime | |
| 开始渲染时间 | StartRenderTime-StartTime | |

4. 系统报表简介

在完成原始指标的加工计算后，后期的数据分析、可视化图表的展示设计就变得非常重要。多维度的数据分析，加上清晰易懂的可视化图表，能够帮助我们更高效地发现、分析和解决问题。



基于笔者的经验，下面的维度可以作为常规性能监控系统的常规设计，尤其是当你的网站用户遍布全球时：

- 基于不同的设备（区分终端），来自手持设备还是 PC 等。
- 基于不同的国家，不同国家中的不同地区等。
- 基于不同类型的用户群体。

然后基于上面不同维度、不同组合的需求，对数据进行聚合计算，并最终用于图表展示。同样也有可借鉴的常规可视化图表设计：

- 性能数据汇总概括。
- 变化趋势。
- 分布区间、直方图等。
- 基于页面的瀑布图绘制（瀑布图绘制可基于最新的 Resource Timing 提供的数据进行绘制）。

在这里有必要强调一点，对数据进行聚合计算时，通常大家第一时间可能会想到使用平均数来统计计算。这会导致一个问题，即平均数非常容易受极大值和极小值的影响，如果只看平均数，会导致我们把访问页面非常慢的那部分用户忽略掉。

比如一个用户打开页面用时 10s，另一个用户打开页面用时 3s，平均耗时为 6s。决策者基于这个结果判断出当前页面性能处于还能接受的状态，但实际上打开页面需要 10s 的问题被忽略了，无法暴露出来。

所以推荐大家使用中位数，可以弥补平均数的不足。用中位数来描述一个页面性能数据的集中趋势，然后结合数据区间，来观察不同区间的分布情况，了解页面真实的性能情况。

比如基于自定义的不同数据区间，定义“非常快”“快”“慢”“非常慢”等区间，来观察用户访问性能快慢的分布情况，从而进行针对性的优化。



4

第 4 章

服务端性能优化

4.1 最大 QPS 推算及验证

影响 QPS（吞吐量）的因素有哪些，一直以来都众说纷纭。大家一般是这么说的：

- QPS 受编程语言的影响。
- QPS 主要受编程模型的影响，比如是不是 coroutine、是不是 NIO、有没有阻塞。
- QPS 主要由业务逻辑决定，业务逻辑越复杂，QPS 越低。
- QPS 受数据结构和算法的影响。
- QPS 受线程数的影响。
- QPS 受系统瓶颈的影响。
- QPS 和 RT 关系非常紧密。
-



以上描述好像都有点对，又有点不对，好像总是“东一点、西一块”，不太完整。接下来要阐述的就是如何通过一些现有的理论和方法来考察系统中可以提高的性能点，将这些“东一点、西一块”的东西总结归纳起来，形成一个体系，用这个体系指导我们更好地对服务端性能进行优化。

在对某电商的活动页面进行优化时，一开始很多人认为 Gzip 不是影响 CPU 的最大因子，直到拿出一一次又一次的实验数据，团队成员才开始慢慢地接受。这里笔者将从另一个角度分析，为什么在某电商活动页面中 Gzip 是影响 CPU 的最大因子。从本节也可以看出，性能优化的相关知识和数据都是体系化的。

4.1.1 RT

公式一：RT = Thread CPU Time + Thread Wait Time

RT (Response Time, 响应时间) 可以简单地理解为系统从输入到输出的时间间隔，系统是指一个网站或者一个其他类型的软件应用，或者指某个设备，比如手机，手机界面也有响应时间。所以 RT 是一个比较广泛的概念，不过在接下来的场景中，RT 都特指互联网应用。服务器端 RT 的含义是指从服务器接收请求到该请求响应的全部数据被发往客户端的时间间隔。客户端 RT 的含义是指从客户端（比如浏览器）发起请求到客户端（比如浏览器）接收该请求响应的全部数据的时间间隔。需要注意的是，服务器端 RT+网络开销≈客户端 RT。也就是说，一个差的网络环境会导致两个 RT 差距悬殊（比如，从俄罗斯到美国的 RT 远大于国内网络环境中的 RT）。

客户端的 RT 直接影响用户体验，要想降低客户端 RT，提升用户体验，必须考虑两点，一个是服务器端的 RT，另一个是网络。对于网络来讲，常见的优化方式有 CDN、AND 和专线，分别适用于不同的场景。

对于服务器端的 RT 来说，主要看服务器端的做法。从公式一中可以看到，要想降低 RT，就要降低 Thread CPU Time 或者 Thread Wait Time。所以接下来着重讨论服务器 RT、Thread CPU Time、Thread Wait Time 等相关知识。

在后面的内容中，将用 CPU Time 替代 Thread CPU Time，用 Wait Time 替代 Thread Wait Time。



4.1.2 单线程 QPS

在上一节中，简单地定义了 RT 由两部分组成，一个是 CPU Time，另一个是 Wait Time。如果系统里只有一个线程或者一个进程（该进程中只有一个线程），那么最大的 QPS 是多少呢？假设 RT 是 199ms（CPU Time 为 19ms，Wait Time 是 180ms），那么 1000ms 以内系统可以接收的最大请求数就是 $1000\text{ms}/(19\text{ms}+180\text{ms}) \approx 5.025$ 。

所以单线程的 QPS 变成了：

$$\text{单线程 QPS} = 1000\text{ms} / \text{RT}$$

4.1.3 最佳线程数

还是从举例开始，延续上面的例子（CPU Time 为 19ms，Wait Time 是 180ms），假设 CPU 的核数是 1。

假设只有一个线程，这个线程在执行某个请求时，CPU 真正花在该线程上的时间就是 CPU Time，可以看作 19ms，那么在整个 RT 生命周期中，还有 180ms 的 Wait Time，CPU 在做什么？在理想情况下，抛开系统层面的问题，可以认为 CPU 在这 180ms 里没做什么，至少对业务来说没做什么。

- 一核的情况

由于每个请求的接收，CPU 只需要工作 19ms，所以在 180ms 的时间内，可以认为系统还可以额外接收 $180\text{ms}/19\text{ms} \approx 9$ 个请求。由于在同步模型中，一个请求需要一个线程来处理，因此，我们需要额外的 9 个线程来处理这些请求。这样，总的线程数就是：

$$(180\text{ms} + 19\text{ms}) / 19\text{ms} \approx 10 \text{ 个}$$

多线程之后，CPU Time 从 19ms 变成了 20ms，这 1ms 的差值代表多线程之后线程上下文切换、GC 等带来的额外开销（如果是 JVM），这里的 1ms 代表一个概数，你也可以把它看作 n 。

- 两核的情况

一核的情况下可以有 10 个线程，那么两核呢？在理想情况下，可以认为最佳线程数为：

$$2 \times (180\text{ms} + 20\text{ms}) / 20\text{ms} = 20 \text{ 个}$$



- CPU 利用率

上面举的都是 CPU 在满载情况下的例子。有时由于某个瓶颈，导致 CPU 得不到有效利用，比如两核的 CPU，因为某个资源，只能各自使用一半的效能，这样总的 CPU 利用率变成了 50%，在这样的情况下，最佳线程数应该是：

$$50\% \times 2 \times (180\text{ms} + 20\text{ms}) / 20\text{ms} = 10 \text{ 个}$$

根据上面的分析，最佳线程数公式如下：

$$\text{最佳线程数} = (\text{RT} / \text{CPU Time}) \times \text{CPU 核数} \times \text{CPU 利用率}$$

当然，最佳线程数公式不是任意推测的，在一些权威著作上都有论述，所以接下来，假设最佳线程数的公式是正确的。

4.1.4 最大 QPS

1. 最大 QPS 公式推导

假设知道最佳线程数，也知道每个线程的 QPS，那么线程数乘以每个线程的 QPS 即这台机器在最佳线程数下的 QPS。所以可以得到如图 4-1 所示的推算过程。

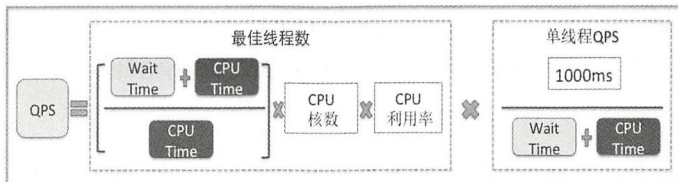


图 4-1

把分子和分母去约数，如图 4-2 所示。

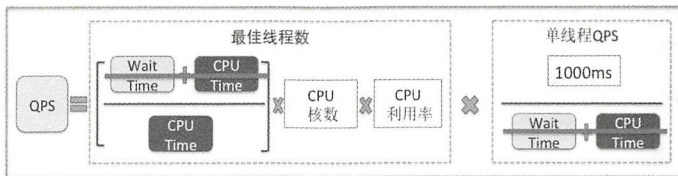


图 4-2

于是公式简化成如图 4-3 所示。

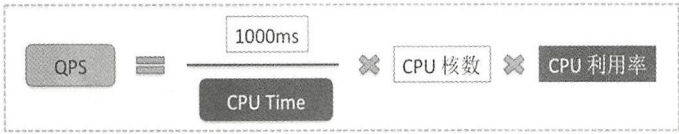


图 4-3

从公式可以看出，决定 QPS 的是 CPU Time、CPU 核数和 CPU 利用率。CPU 核数是由硬件决定的，很难操纵，但是 CPU Time 和 CPU 利用率与代码息息相关。

虽然宏观上是正确的，但是推算的过程中还是有一点小小的不完美，因为多线程下的 CPU Time（比如高并发下的 GC 次数增加消耗更多的 CPU Time、线程上下文切换等）和单线程下的 CPU Time 是不一样的，所以会导致推算出来的 QPS 和实际的 QPS 有误差。

尤其是在同步模型下的相同业务逻辑中，单线程时的 CPU Time 肯定会比大量多线程的 CPU Time 小，但是对于异步模型来说，切换的开销会变小，对此后面将会详细描述。

既然决定 QPS 的是 CPU Time 和 CPU 核数，那么这两个因子又是由什么决定的呢？

2. CPU Time

CPU Time 不只是业务逻辑所消耗的 CPU 时间，而是一次请求中所有环节上消耗的 CPU 时间之和。比如在 Web 应用中，一个请求过来的 HTTP 的解析所消耗的 CPU 时间，是 CPU Time 的一部分，另外，这个请求中请求 RPC 的 encode 和 decode 所消耗的 CPU 时间也是 CPU Time 的一部分。

那么 CPU Time 是由哪些因素决定的呢？两个关键字：数据结构+算法。举几个例子：

- 均摊问题。
- Hash 问题。
- 排序和查找问题。
- 状态机问题。
- 序列化问题。

3. CPU 利用率

CPU 利用率不高的情况是时常发生的，以下因素都会影响 CPU 利用率，从而影响系统可以支持的最大 QPS。



(1) I/O 能力。

- 磁盘 I/O
- 网络 I/O
 - 带宽，比如某大促压力测试时，由于某个应用放在 Tair 中的数据量大，导致 Tair 的机器网卡跑满。
 - 网络链路，还是这次大促，借用了其他核心交换机下的机器，导致客户端 RT 明显增加。

(2) 数据库连接池（并发能力 = $\text{PoolWaitTime}^{\text{①}} / \text{RT}(\text{client}) \times \text{PoolSize}^{\text{②}}$ ）。

(3) 内存不足，GC 大量占用 CPU，导致给业务逻辑使用的 CPU 利用率下降，而且 GC 时还满足 Amdahl 定律锁定义的场景。

(4) 共享资源的竞争，比如各种锁策略（读写锁、锁分离等），各种阻塞队列，等等。

(5) 所依赖的其他后端服务 QPS 低造成瓶颈。

(6) 线程数或者进程数，乃至编程模型（同步模型、异步模型，某些场景适合同步模型，某些场景适合异步模型）。

在压力测试的过程中，出现最多的是网络 I/O 层面的问题，GC 大量占用 CPU Time 之类的问题也经常出现。

4. CPU 核数，Amdahl 定律，Gustafson 定律

1) Amdahl 定律（安达尔定律）

Amdahl 定律是用来描述可伸缩性的，什么是可伸缩性？按照权威资料解释：

可伸缩性是指，当增加计算资源的时候，如 CPU、内存、带宽等，QPS 能够相应地进行改进。

那么 Amdahl 定律是如何来描述可伸缩性的呢？

Amdahl 在自己的论文中指出，可伸缩性是指在一个系统中，基于可并行化和串行化的组件

① PoolWaitTime，连接池等待时间。

② PoolSize，连接池大小。



各自所占的比例，当程序获得额外的计算资源（如 CPU 或者内存等）时，系统理论上能够获得的加速值（QPS 或者其他指标可以翻几倍）。用一个公式来表示，如果 F 表示必须串行化执行的比例，那么在一个 N 核处理器的机器中，加速：

$$\text{Speedup} \leq \frac{1}{F + \frac{1-F}{N}}$$

这个公式代表的意义是比较广泛的，在项目管理中也有一句类似的话：

一个女人生一个孩子，需要 9 个月，但是永远不可能让 9 个女人在一个月内生一个孩子。

我们拿公式套用一下，这里 $F=100\%$ ，9 个女人表示 $N=9$ ，于是就有 $1/(100\% + (1-100\%)/9) = 1$ ，所以 9 个女人的加速比为 1，等于没加速。

这里要提醒大家，这个公式描述的是，在增加资源的情况下系统的加速比，而不是在资源不变的情况下优化数据结构和算法之后带来的提升。优化数据结构和算法带来的提升要看前文中的最大 QPS 公式。不过这两个公式也不是完全没有联系的，在增加资源的情况下，它们的联系还是比较紧密的。

2) Gustafson 定律（古斯塔夫森定律）

这个定律名字很长，它是 Amdahl 定律的补充：

$$S(P) = P - \alpha \cdot (P - 1)$$

P 是处理器个数， α 是串行时间占总执行时间的比例。

生孩子的案例再次套上这个定律， P 为女人个数，等于 9，串行比例是 100%。Speedup = $9 - 100\% \times (9 - 1) = 1$ ，也就是 9 个女人是无法在一个月把孩子生出来的……

两个定律有关系吗？有，它们是相辅相成的关系。前者从串行和并行执行时间的角度来推导，后者从串行和并行的计算量角度来推导，不管哪个角度，最终的结果是一样的。

3) CPU 核数和 Amdahl 定律的关系

通过最大 QPS 公式，笔者发现，在 CPU Time 和 CPU 利用率不变的情况下，核数越多，QPS 就越大。比如核数从 1 到 4，在 CPU Time 和 CPU 利用率不变的情况下，加速比应该是 4，所以 QPS 应该增加 4 倍。

这是资源增加（CPU 核数增加）的情况下的加速比，也可以通过 Amdahl 定律来衡量，考



考虑串行和并行的比例在增加资源的情况下是否会改变。也就是要考虑在 N 增加的情况下， F 受哪些因素的影响：

$$\text{Speedup} \leq \frac{1}{F + \frac{1-F}{N}}$$

只要 F 大于 0，最大 QPS 就不会翻 4 倍。

一个公式说要增加 4 倍，一个定理说没有 4 倍，互相矛盾？

其实事情是这样的，通过最大 QPS 公式，我们得知，如果 CPU Time 和 CPU 利用率不变，核数从 1 增加到 4，QPS 会相应地增加 4 倍。但是在实际情况下，当核数增加时，CPU Time 和 CPU 利用率大部分时候是变化的，所以前面的假设不成立，即一般场景下最大 QPS 不能增加 4 倍。

而 Amdahl 定律中的 N 变化时， F 也可能会变化，即一般场景下，最大 QPS 并不能增加 4 倍，所以不矛盾，相反它们是相辅相成的。这里一定要注意，这里说的是一般场景，如果你的场景完全没有串行（程序没有串行，系统没有串行，上下文切换没有串行，什么串行都没有），那么理论上还是可以增加 4 倍的。

为什么增加计算资源时，最大 QPS 公式中的 CPU Time 和 CPU 利用率会变化， F 也会变化呢？我们可以从宏观上分析一下，增加计算资源时，达到满载：

- QPS 会更高，单位时间内产生的对象会更多。在同等条件下，minor GC 被触发的次数增加，还有些场景发生过对象多到响应没返回它们就进了“老年代”，从而 full GC 被触发。宏观上，这是属于串行的部分，对于 Amdahl 公式来说 F 会受到影响，对于最大 QPS 公式来说，CPU Time 和 CPU 利用率也受到影响。
- 在同步模型下大量的线程在完成一次请求中，上下文被切换的次数大大增加。
- 尤其是在有串行模块的时候，串行的执行和等待时间增加， F 会变化，某些场景下 CPU 利用率也达不到理想效果，这取决于你的代码。这也是要做锁分离、为什么要缩小同步块的原因。当然还有锁自身的优化，比如偏向、自旋、读写分离等技术，都是为了不断地减少 Amdahl 定律中的 F ，也是为了减少 CPU Time（锁本身的优化），提高 CPU 利用率（使用锁的方法的优化）。
 - 锁本身的优化最为津津乐道的是自旋、偏向、自适应，这些知识点请看网上的 `synchronized` 分析，还有 `reentrantLock` 的代码及 AQS 论文。
 - 使用锁的优化方法最常见的是缩小锁区间、锁分离、无锁化、`volatile`。



所以在增加计算资源时，更高的并发产生，会引起最大 QPS 公式中两个参数的变化，也会引起 Amdahl 定律中 F 值的变化，同时公式和定律变化的趋势是相同的。Amdahl 定律是得到广泛认可的，也是得到数据验证的。最大 QPS 公式好像没有人验证过，这里引用一个比较有名的测试结果，如图 4-4 所示。

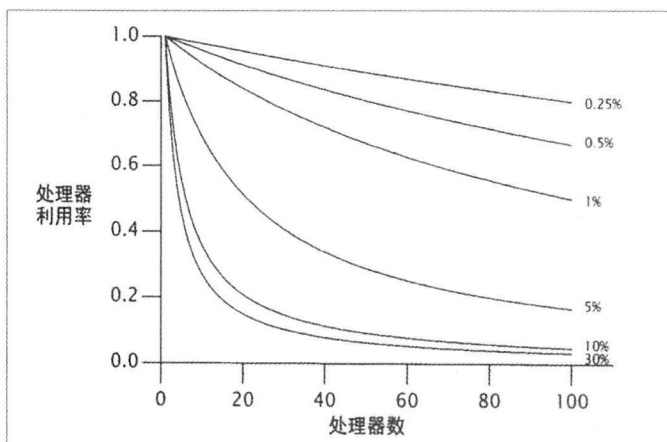


图 4-4

从图 4-4 中可以看到：

- 当计算资源（处理器数量）增加时，在串行部分比例不变的情况下，CPU 利用率下降。
- 当计算资源（处理器数量）增加时，串行占的比例越大，CPU 利用率下降得越多。

4.1.5 实验数据验证公式

截至目前，笔者都在讲公式，但是这个公式是不是正确，应该如何测量？能否做到精确测量，还是做一个概要测试即可？

注意：在这个讲解中，CPU Time 包含了操作系统对 CPU 的消耗，比如进程、线程调度等。

1. 数据准备

之前在某电商活动页面的优化过程中，笔者做了大量测试，由于测试中使用了 localhost 方式，所以 PHP 进程在 I/O 上的 Wait Time 是非常小的。接下来，由于最佳线程数接近 CPU 核数，所以在两核的机器上使用了 10 个 PHP 进程，客户端发起了 10 个并发请求，可以认为这是在最佳线程数下（最佳线程数在一个区间里，在这个区间里 QPS 总体变化不大，笔者也用 5、15 个并发，QPS 值基本相同）得出的大量结果，接下来就分析一下这些测试结果。



1) 测试出来的 QPS（表 4-1）

表 4-1

| 原始页面大小 | 压缩后的大小 | 优化前QPS | 优化后QPS | 优化前RT | 优化后RT |
|--------|--------|--------|--------|---------|-------|
| 92KB | 17KB | 164 | 2024 | 60.7ms | 4.9ms |
| 138KB | 8.7KB | 143 | 1859 | 69.8ms | 3.3ms |
| 182KB | 11.4KB | 121 | 2083 | 82.3ms | 4.8ms |
| 248KB | 32KB | 77 | 1977 | 129.6ms | 5.0ms |
| 295KB | 34.4KB | 70 | 1722 | 141.1ms | 5.8ms |

只需要关注表 4-1 中各页面优化前的 QPS 和优化后的 QPS 即可。

2) CPU 利用率

由于 Apache Bench 和 PHP 部署在同一台机器上，所以 CPU 利用率应该减去 Apache Bench 的 CPU 资源消耗。根据观察，优化前 Apache Bench 的 CPU 消耗在 1.7%到 2%之间，优化后 Apache Bench 的 CPU 资源消耗在 20%左右。为什么优化前后有这么大的差距呢？因为优化后响应能够及时返回，所以导致 Apache Bench 使用的 CPU 资源多了。

在接下来的计算中，笔者将优化前的 CPU 利用率设置为 98%，优化后的 CPU 利用率设置为 80%。

3) CPU Time 计算公式

根据 QPS 的计算方法，把 QPS 挪到右边的分母中，CPU Time 移到等号左边，就会得到如图 4-5 所示的公式。

$$\text{CPU Time} = \frac{1000}{\text{QPS}} \times \text{CPU 核数} \times \text{CPU 利用率}$$

图 4-5

4) CPU Time 计算示例

根据上面列出的 3 点（CPU 利用率、QPS 和 CPU 核数），可以推算出优化前和优化后的 CPU Time，推算方法如表 4-2 所示。

2. 计算得到 CPU Time

根据上面表格的计算方法，利用 QPS 计算出各页面理论上的 CPU Time，计算后的结果如



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

表 4-3 所示。

表 4-2

| 原始页面92KB | 计算公式 |
|---------------|--|
| 优化前CPU Time计算 | $1000 / 164 \times 2 \times 0.98 = 12\text{ms}$ |
| 优化后CPU Time计算 | $1000 / 2024 \times 2 \times 0.8 = 0.8\text{ms}$ |

表 4-3

| 原始页面大小 | 压缩后的大小 | 优化前QPS | 优化后QPS | 优化前CPU Time | 优化后CPU Time |
|--------|--------|--------|--------|-------------|-------------|
| 92KB | 17KB | 164 | 2024 | 12ms | 0.8ms |
| 138KB | 8.7KB | 143 | 1859 | 13.7ms | 0.86ms |
| 182KB | 11.4KB | 121 | 2083 | 16.2ms | 0.77ms |
| 248KB | 32KB | 77 | 1977 | 25.5ms | 0.81ms |
| 295KB | 34.4KB | 70 | 1722 | 28ms | 0.93ms |

请大家着重注意优化前的 CPU Time 及优化后的 CPU Time，接下来将两个值做减法，然后和实测程序中 Gzip 的 CPU Time 进行对比。

3. 实测 CPU Time

1) 5 个页面的 Gzip 所消耗的 CPU Time

实测 5 个页面做 Gzip 所消耗的时间，然后跟公式计算出来的 CPU Time 做一个对比，如表 4-4 所示。

表 4-4

| 原始页面大小 | CPU Time公式差值(表4-3的CPU Time 差值) | Gzip CPU Time测量值 (10次平均值) | 差 值 |
|--------|--------------------------------|---------------------------|-------|
| 92KB | 11.2ms | 8ms | 3.2ms |
| 138KB | 12.8ms | 7ms | 5.8ms |
| 182KB | 15.4ms | 9ms | 6.4ms |
| 248KB | 24.7ms | 21ms | 3.0ms |
| 295KB | 27.1ms | 23ms | 4.1ms |

可以看到，计算出来的 CPU Time 值要比测量出来的 Gzip 的时间多几毫秒，这是为什么呢？

因为在优化前，有两个消耗 CPU Time 的阶段，一个是执行 PHP 代码时，另一个是执行





Gzip 时。而优化后，整个逻辑变成了从缓存获取数据后直接返回，只有非常少量的 PHP 代码在消耗 CPU Time（10 行以内）。

2) PHP 页面执行消耗的 CPU Time

大体上可以认为：

优化前的 CPU Time - 优化后的 CPU Time = GZIP CPU Time + 全页面 PHP 代码的 CPU Time

在实验中，一开始只统计了 Gzip 本身的消耗，而在 PHP 文件中 PHP 代码执行的时间并没有包含在内，所以两者差距比较大。于是，笔者单独统计了 5 个页面的 PHP 代码的执行时间，发现文件中 PHP 代码执行的时间为 3~6ms。实际测量的 Gzip CPU Time 加上 3~6ms 的 PHP 代码执行时间，和使用公式计算出来的 CPU Time 基本吻合。

根据上面的计算和测量结果，笔者发现 Gzip 的 CPU Time 消耗加上 PHP 代码的 CPU Time 消耗，与公式测量出来的总的 CPU Time 消耗非常接近，误差为 1~2ms。考虑到 CPU Time 测量是单线程测量，而压力测试 QPS 是并发情况下（会多出进程切换的开销和 GC 等的开销），笔者认为这点误差是合理的，测试结果表明公式在宏观上是正确的。

4.1.6 压力测试最佳线程数和 QPS 的临界点

前面讲到了公式的推导，并在一个固定的条件下验证了公式在该场景下的正确性。

假设在一个 thread-per-client 的场景，有一个 Ajax 请求，这个请求返回一个 Json 字符串，每个请求的 CPU Time 为 1ms，Wait Time 为 300ms（比如读写 Socket 和线程调度的等待开销）。那么最佳线程数是 $(300+1/1) \times 4 \times 100\% = 1204$ 。尤其在广域网上，Wait Time=300ms 是正常的数值。在国际环境下，300ms 就更加常见了。这意味着如果是 4 核的机器，需要 1204 个线程，如果是 8 核的机器则需要 2408 个线程。实际上，有些 HTTP 服务的 CPU Time 是远小于 1ms 的，比如上面的场景中将页面压缩并缓存起来之后，CPU Time 基本为 0.8ms，如果 Wait Time 还是 300ms，那么需要数以千计的线程啊！

当线程数不断增加的时候，到达某个临界点之后对系统就开始产生负面影响了。

(1) 大量线程上下文切换的开销，引起 CPU Time 的增加及 QPS 的减少。所以，有时候还没有达到最佳线程数，而 QPS 已经开始略微下降了。因为 CPU Time 发生变化、线程多了之后，调度引起的 CPU Time 提升的百分比和 QPS 下降的百分比成正比（详见 QPS 公式），上下文切换带来的开销如下。





- 上下文切换（微秒级别）。
- JVM 本身的开销。
- CPU Cache 加载。

（2）线程的栈空间会占用大量内存，假设每个线程的栈空间是 1MB，这么多线程就要占据数 GB 的内存。

（3）在 CPU Time 不变的情况下，因为线程上下文切换和操作系统想尽力为线程在宏观上平均分配时间片的行为，导致每个线程的 Wait Time 都增加了，于是每个请求的 RT 也增加了，最终导致用户体验下降。

可以用一张简单的示意图（图 4-6）来表示临界点的概念。

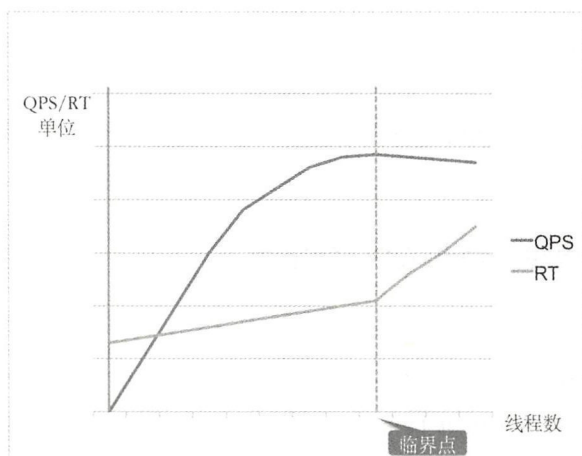


图 4-6

由于线程数增加超过某个临界点会影响 CPU Time、QPS 和 RT，所以很难精确测量高并发下的 CPU Time，它随着机器硬件、操作系统、线程数等因素不断变化。笔者能做的就是压力测试 QPS，并在压力测试的过程中调整线程数，使 QPS 达到临界点，这个临界点是 QPS 的一个峰值点。这个峰值点的线程数即当前系统的最佳线程数，当然如果这个时候 CPU 利用率没有达到 100%，那么证明系统中可能存在瓶颈，应该在找到并处理瓶颈之后继续压力测试，并且重新找到这个临界点。

当数据结构发生改变、算法改进或者业务逻辑发生改变时，最佳线程数有可能会跟着变化。

在本节举的例子中，在 CPU Time 下降到 1ms 左右而 Wait Time 需要数百毫秒的场景下，我们需要很多线程。但是当达到这个线程数的时候，有可能早就达到了临界点。所以系统整体





已经不是最健康的状态了，但是现有的编程模型已经阻碍了我们前进，那么应该怎么办呢？为使某个系统达到最优状态，我们来看看编程中的同步模型和异步模型问题，以及为什么异步模型只需要这么少的线程，是不是公式在异步模型下失效了。

4.2 同步模型与异步模型

通常，Web 应用使用同步模型开发最简单，某些特别的应用可以使用异步模型，比如可以使用 Netty+coroutine 模型或 continuation+Kilim 的 coroutine 等。那么异步模型是不是和公式相悖？在不同场景下应该选择什么样的模型？

4.2.1 同步模型

1. 同步模型简介

关于什么是同步模型，拿 Web 开发中的 thread-per-client 举一个经常碰到的例子：

- (1) 浏览器发起 HTTP 请求，servlet 容器接收请求。
- (2) servlet 容器解析请求，并按照业务逻辑请求 remote Cache 中的内容（Tair 等）。
- (3) servlet 容器拿到 Cache 的返回结果，并进行逻辑运算。
- (4) servlet 容器将计算结果返回。

假设用 Tomcat 作为 servlet 容器，当一个请求过来时（由于流程较为简单，直接用文字描述流程）：

- (1) Tomcat 中监听 8080 端口的主线程接收了一个 socket。
- (2) 主线程把 socket 交给 Tomcat 线程池中的某个 work 线程。
- (3) 线程阻塞式读取 socket 中的数据，并解析 HTTP，组装成 request 对象。
- (4) 线程获取 request 对象中的某个数据作为 key。
- (5) 根据上一步获取的 key 请求 Tair，线程会进入阻塞状态，等 Tair 的 value 返回。
- (6) value 返回，线程被唤醒，将 value 返回给 response。
- (7) 返回结果以阻塞写的方式写入 socket 的 OutputStream。





(8) Tomcat 的 work 线程回池，准备处理下一个请求。

可以看到，线程分别在 (3)、(4)、(5)、(6)、(7) 步进入 block 状态，尤其是第 (3) 步和第 (7) 步，如果是在广域网上，这个线程将产生大量的 Wait Time。

这就是传统的同步模型，同步模型基本基于 BIO。但是上例中的请求 Tair 并不是 BIO，而是 NIO，虽然在 I/O 上是非阻塞的，但是线程依然进入了阻塞状态，不过是阻塞在 countdownlatch 或者 blockingqueue 上。对整个模型来说，这依然是一个同步模型。

2. 同步模型的优缺点

1) 同步模型的优点

同步模型编程简单，符合人类的思维方式，维护成本也低，尤其当有复杂的业务逻辑时，对线程数的要求会下降到几百个甚至几十个，此时使用同步模型是一个不错的选择。在极端情况下，尤其对于一个 CPU 强密集型的应用，一个请求过来的时候，有较强的业务逻辑，造成了大量的 CPU Time，而 Wait Time 却只占 RT 很小的比例。这个时候，通过公式算出的线程数将会维持在接近 CPU 核数的数量级上，使用同步模型是非常合适的。在存在大量业务并且调用和依赖比较复杂的情况下，使用同步模型开发和维护的成本都比异步模型要低很多。这里的开发和维护成本包含代码编写成本和理解成本，以及故障处理的成本等。

2) 同步模型的缺点

在某些场景下无法达到最优的 QPS，按照前面举的例子（假设 CPU Time 在 1ms 以内，Wait Time 在 100ms 以上），为了让 QPS 最大化，需要设置数千个线程，但是由于线程数多，导致 CPU Time 又增加了（线程上下文切换、虚拟机的开销），于是 QPS 和理想的最大值比又会下降。在这种场景下，使用同步模型就比较吃力，举一个现实点的例子，比如 DNS 中的智能路由，本来一个请求的 CPU Time 就大大小于 1ms，如果使用同步模型，Wait Time 中会包含大量的 I/O Wait Time，于是需要数以千计的线程，这是程序员们无法接受的。

其他优缺点与本书主题无关，不再赘述。

4.2.2 异步模型

1. 异步模型的推导

全异步模型的特点是可以支持大量的链接，并且在业务执行过程中没有任何地方有 I/O 阻塞，这个技术特性基于 NIO 或者 AIO 才能实现。比如，拿前面的场景举例，但是不用 Tomcat 了，笔者使用 Jetty 的 continuation 实现相同的功能，流程如图 4-7 所示。



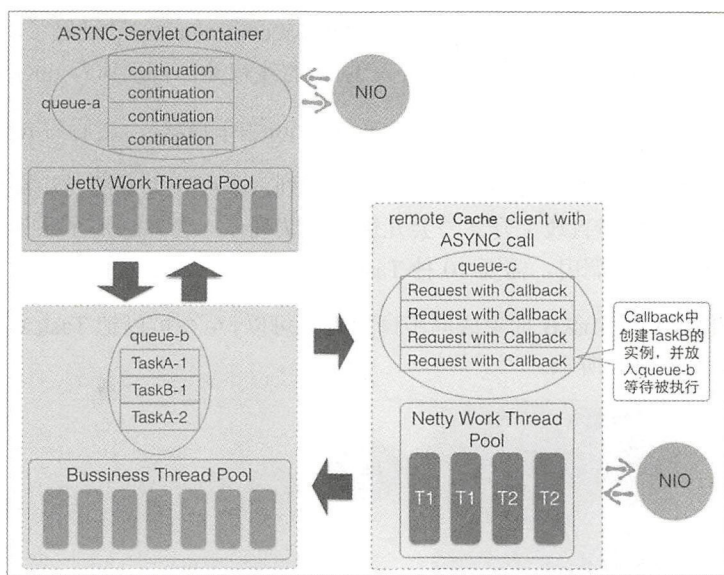


图 4-7

图 4-7 中的几个关键点如下：

- continuation 代表一个连接，它包含了 request 对象和 response 对象，拿着它走遍任何一个队列都不怕，不管执行到哪里，最终依然知道当前正在处理的数据应该返回给哪个请求链接。
- remote Cache client 必须支持异步调用，数据返回由负责 I/O 的线程把 TaskB 放入业务执行队列，等待被业务线程执行。
- TaskB 由 TaskA 创建，无须 remote Cache client 创建。

简单来说，图 4-7 的处理流程为：

- (1) Jetty 线程获取 continuation（包含 request、response、connector），并创建 TaskA。
- (2) TaskA 被业务线程池中的线程执行。
- (3) 在 TaskA 的代码中异步调用 memcached client。

(4) memcached client 中的线程获取结果之后，将之前 TaskA 创建好的 TaskB 再次放入业务线程池执行。

(5) TaskB 根据获取的 Cache 计算业务逻辑，获取处理结果（业务线程执行），并放入 continuation。





(6) TaskB 调用 continuation 的方法，告知 Jetty 该 continuation 已经好了，可以写 response 了（其实也是把 continuation 相关的上下文放入 Jetty 的线程池，再次执行 doGet）。

(7) Jetty 线程再次进入 doGet，并得到 continuation 中的数据，然后写 response。

由图 4-7 和上述流程可知，异步模型严重依赖队列，并存在大量的异步回调，如果硬编码这些回调，代码的复杂度呈直线上升（一会儿进这个队列，一会儿进那个队列）。试想如果一个请求需要 10 种 9 个远程调用，需要 10 种 Task，代码会变成什么样子？

是否可以将 TaskA 和 TaskB 抽象成同步调用呢？可以，至少可以把 TaskA 和 TaskB 合并成一个 TaskAB 类，如图 4-8 所示。

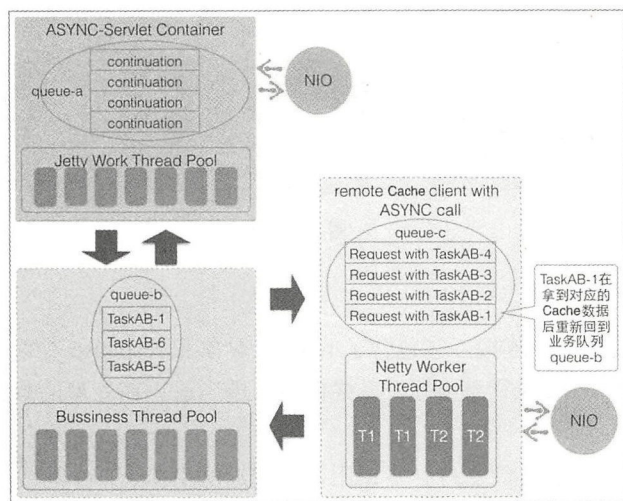


图 4-8

在图 4-8 中，可以看到 TaskA 和 TaskB 不见了，变成了一个 TaskAB 类，而且这个 TaskAB 类会被放进业务线程池两次，第一次是在执行业务代码并异步调用 Cache client 时，第二次是在获取 Cache client 的返回结果时。

两次执行 TaskAB，TaskAB 中就需要用 if else 的逻辑处理了，示例代码如下。

```
public class TaskAB extends Task{

    private static final Log logger = LogFactory.getLog(DownloadTask.class);

    private RoutingService routingService;
    private DownloadDO downloadDO;
    private final Continuation jjcontinuation;
```





```
private CacheManager memcachedManager;
private final HashMap<String, CASValue<Object>> result = new
HashMap<String, CASValue<Object>>();
private final ExecutorService pool;

void execute() {
    try {
        if (result.size() == 0) {
            //第一入口
            final String key = XXXUtils.geneItemCacheKey(downloadDO.
getFd(), downloadDO.getOrg());
            memcachedManager.gets(key, new MemcachedCallBack
<Future<CASValue<Object>>>>() {
                private Future<CASValue<Object>> o;

                public void setAttachment(Future<CASValue<
Object>> o) {
                    this.o = o;
                }
                /*这里模仿了协程的 resume, 先把 Cache value 放到 TaskAB
中, 然后调用 resume 将其放入线程池队列。*/
                public void resume() {
                    try {
                        result.put(key, o.get());
                    } catch (Exception e) {
                        //日志
                    }

                    pool.execute(DownloadTask.this);
                }

                public Future<CASValue<Object>> getAttachment() {
                    return o;
                }
            });
        } else {
            //第二入口
            try {
                final String key = XXXUtils.geneItemCacheKey
(downloadDO.getFd(), downloadDO.getOrg());
                CASValue<Object> rs = result.get(key);
                FileLocation fileLocation = routingService.getFileLocation(downloadDO, rs);
```





```
        jjcontinuation.setAttribute(XXXConstants.
CONTINUATION_ATTR, fileLocation);
    } finally {
        /*通知 Jetty, 让 Jetty 再为该 request 进入 doGet 方法, 原理和
resume 类似*/
        jjcontinuation.resume();
    }
}
} catch (Throwable e) {

    getLog().error("Business logic error, wait MQ sotimeout", e);
    jjcontinuation.setAttribute(CDCCConstants.CONTINUATION_ATTR, e);
}
}
```

这段代码的核心是：

这里笔者模仿了协程的 resume，先把 Cache value 放到 TaskAB 中，再将其放入线程池队列，让它能够再次被执行，并进入 execute 中的 else 方法。

同样，也是利用 Jetty 的 continuation 的 resume 方法，可以让 Jetty 为该 continuation 中包含的 request 和 response 再次进入 doGet 方法。

如果异步场景较少，使用该方式无伤大雅，反而是最容易实现的，不需要引入其他框架，只需要 if elseif 即可。但是如果这样的场景很多，这么写就有点麻烦了。如果有 9 次远程的异步调用，则需要 10 次左右的 if elseif。于是就有人想到把 if else 这样的东西隐藏起来。

不就是 9 次异步调用吗？如果把 TaskAB 看作一个线程呢？9 次调用不就是 9 次 wait（等待）和 9 次 notify（通知）吗？然后在 TaskAB 上创建一个共享的变量，比如叫作 mailbox，异步请求的响应回来之后把结果放到 mailbox 中，然后 notify 一下 TaskAB（比如把 TaskAB 再次放入线程池中执行），这样 TaskAB 不就获取了第一个异步请求的结果了吗？后面遇到异步的地方就 wait，等数据进入 mailbox 就再被 notify，是不是也可以？

事实上，我们无法将 TaskAB 看作一个线程，因为我们的场景可能需要很多线程，而这么多线程会导致 QPS 与理论最大值的差距很大，所以有人创造出了“协程”的概念。

协程也可以对 if elseif 等跳转进行抽象和封装，从代码的角度看与 wait/notify 类似。但是实际上 TaskAB 是一个普通的对象，它还要丢进线程池的阻塞队列中等待被执行，只不过这些细节都被协程框架封装了，而且也不叫 wait/notify 了，改成 pause/resume 或者 suspend/resume 了。



2. 延伸阅读：Kilim 简介

在 Java 中，也有一个比较出名的协程框架叫作 Kilim。使用 Kilim 可以将 if elseif 用 pause 和 resume 进行替换，不过这只是在 Java 源代码里的替换，当源代码被编译成了字节码，其实又回到了 if elseif 之类的跳转模式。这也是笔者在前面介绍 TaskAB 和 if elseif 的原因。

Kilim 可以把 if elseif 用 pause 和 resume 替换，它是怎么做的呢？说来也简单，字节码增强 + 整套类线程的机制（线程调度变成协程调度，线程执行队列和等待队列变成协程的执行队列和等待队列等，类似的概念，不同层次的实现）。字节码增强就是分析代码中的 Kilim 关键字，并用 if 之类的 Java 关键字替换，以生成字节码。类线程机制是什么呢？请看图 4-9。

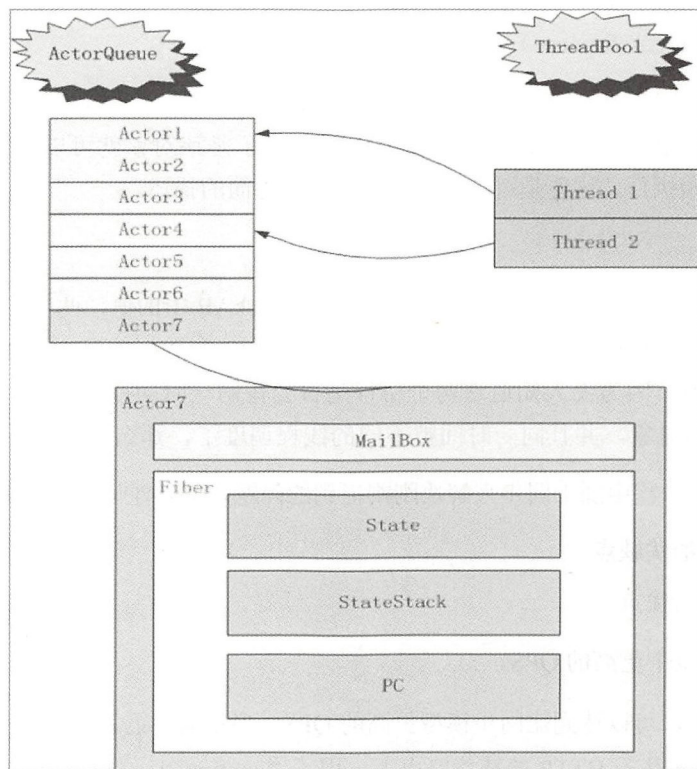


图 4-9

可以看到 TaskAB 其实就是协程，会放到一个队列中，每个协程都有一个 mailbox，有一个 fiber 对象记录该协程执行到了什么地方，fiber 中存储着第 2 次进入 TaskAB 时，应该跳到哪个 elseif 或者哪个 switch 之类的信息。fiber 是一个非常重要的概念。

当然，其中的细节很多，各种状态判定、跳转，也不是只有 if elseif，也有 switch case 的



跳转，反正就是跳来跳去，对于应用者来说，了解基本原理有助于技术选型。

如果异步请求不多，也没必要用 Kilim，直接自己硬编码一个 XXXTask 即可。

这样请求模型从同步的 `client-per-thread` 变成了 `client-per-coroutine`。协程中上下文的切换要比线程上下文的切换省事多了，仅是出入队列而已。

3. 异步模型的总结

1) 暂停与继续

对于异步模型的特点，笔者用两个词来描述，相信仔细看完上述分析过程后对这两个词将不再陌生：`suspend`（或者 `pause`）和 `resume`。

执行某个协程对象时，在任何异步的地方，调用异步方法，并 `suspend`（或者 `pause`），然后等待异步方法的数据返回后调用该对象的 `resume`，于是该对象就可以在它 `suspend`（或者 `pause`）的地方开始执行。这便是对全异步模型最化繁为简的描述。

2) 小忠告

千万别在上节的一个 TaskAB 中去改变另外一个 TaskAB 中的值，两个 Task 之间不应该有数据共享。推导到协程概念上，就是两个协程对象不应该有数据共享，更不能有两个协程修改相同的数据行为。因为没人知道这两个协程是否会在同一时间被不同的线程调度，一旦两个协程修改相同的对象，并且同一时间被不同的线程调用了，那么就麻烦了。

也许有人想在协程中加上同步来解决刚刚提到的问题，笔者建议不要这么做。

4. 异步模型的优缺点

1) 异步模型的优点

(1) 特定场景下更高的 QPS。

在特定场景下，可以达到比同步模型更高的 QPS。多年前，笔者设计并实现过大型视频网站的 CDN 系统，在基于 HTTP 的智能路由上，用了“`continuation+coroutine`”的思想，并改造了 `memcached client` 使之支持异步模型。仅 JVM 中 6 个线程在两核的 PC 上可以跑 6000 的 QPS（Jetty 的读写 I/O 线程数是 6 个，actor 的线程数是 2 个，`memcached` 的线程数是 2 个，`jmeter` 也跑在这台机器上，CPU 利用率当时没有记录），这个例子不是为了说明 QPS 有多高，因为 QPS 高低是由 CPU Time 和 CPU 利用率决定的，只是为了说明改成异步模型之后，处理业务逻辑线程的 I/O Wait Time 减少了，所以只要少量的线程就可以做大量的事情，而如果用同步模型，



线程的数量会导致提前达到临界点，这个临界点的 QPS 距理论最高 QPS 还有一段距离。

(2) 线程数容易固定，导致系统不易过载，高压时只是队列长度增加。

为什么这样说？因为在同步模型下，要为每个应用都设置接近临界点的线程数，这是要花费大量时间来测试的，而这个投入很多时候产出并没有那么高。这时我们往往会设置一个较大的线程数，比如 400，而对于一些应用来说，这 400 个还没有到达临界点，在突然的高压下系统性能得不到充分的发挥。对于另外一些系统来说，已经超过了临界点，高压时系统的 QPS 下降，而 RT 上升。在异步模型下，线程数的设置较为简单，在突然的高压下，虽然 RT 也有上升（因为请求在队列中等待处理需要时间），但是可以省掉同步模型中线程不断切换带来的开销，而系统的最高 QPS 基本不会受到影响。异步系统中的线程将会有条不紊地继续工作。

(3) 限流。

可以通过队列长度来限流。不做好流控，会造成大量请求堆积，如果得不到及时处理，可能导致 full GC 频率加快，这是一个特点。

2) 异步模型的缺点

- 异步模型相对同步模型而言较为复杂，存在大量的异步回调，虽然 Kilim 等框架可以在一定程度上解决代码开发和维护成本问题，但是如果交互异常复杂，存在大量异步交互的场景，尤其是业务逻辑经常变更的 Web 系统，即使使用协程，开发成本和维护成本及稳定性依旧是不小的挑战。
- 即使使用 coroutine 框架，也是有一些学习和理解成本的。
- Thread Local 要视场景使用，使用前一定要把其源代码看一遍。
- 如果想在一条节点很多的链路上都使用异步，那么对于 RPC 框架来说，最好支持连接复用，RPC 协议中需要携带 sequence ID（非必须，也许这是最佳的选择），RPC 框架还得支持回调。在定义接口时也必须考虑回调的问题，服务化接口也要实现异步，架构复杂度大大增加了，到后面会无法控制。

5. 建议的异步模型

建议使用 Jetty 的 NIO selector，并且使用其 continuation，配合 Kilim 或者硬编码异步回调，同时使用 memcached 或者 Tair 之类的分布式 Cache 的异步接口（如没有需自行改造）。这套方案是比较成熟的，而且是可以做到全异步的，即所有的环节都异步。有些应用是半异步的，比如 continuation 和 coroutine 采用了异步，但是在调用后面的依赖时使用同步调用，于是就



需要一个大的线程池来请求服务，如果线程池数量减少，那么 CPU 利用率可能就会下降，线程数超过某个临界点，又会影响最高 QPS。所以如果选择走异步这条路，请尽量考虑全异步方案。

当然，如果不想使用 Jetty 的 continuation，要使用 Netty，那也是可以的。而且如果对外提供的协议不是 HTTP 或者 WebSocket，那么使用 Netty 来实现自定义协议和 continuation 的功能可能是最佳选择。

4.2.3 为什么异步模型需要的线程数少

1. 业务线程数减少的原因

为什么同步模型需要那么多线程，异步模型只需要几个线程呢？这主要归因于非阻塞式 I/O。在同步模型下，线程必须阻塞在 I/O 上面，阻塞在 I/O 上的时间是要计入线程的 Wait Time 的，所以在同步模型转化为异步模型之后，业务线程在 I/O 上阻塞的时间大大减少，根据最佳线程数公式：

$$\text{最佳线程数} = ((\text{Wait Time} + \text{CPU Time}) / \text{CPU Time}) \times \text{CPU 核数} \times \text{CPU 利用率}$$

一旦 Wait Time 逼近 0（只是逼近，因为程序中肯定还存在串行的地方，比如 GC 等），那么最佳线程数可以近似看作：

$$\text{最佳线程数} = (\text{CPU Time} / \text{CPU Time}) \times \text{CPU 核数} \times \text{CPU 利用率}$$

所以这可以视作：全异步模型中线程数=CPU 核数×CPU 利用率。

2. I/O 线程

自从业务线程不做 I/O 相关的工作了，它们觉得自己干活更有力了，激情也更高了，但是 I/O 的活也总得有人干，比如读写浏览器发送过来的数据，需要 Jetty 的 work thread；读写 Tair 的操作，需要 mina 或者 Netty 的 work thread。这些 thread 该如何设置呢？比如 Netty 的 thread，默认情况下设置成了“CPU 核数+1”。根据某些权威资料，这么做的理由是：即使当 CPU 密集型的线程因为页缺失或者其他故障导致该线程暂停，这个额外的线程也可以保障 CPU 的时钟周期不被浪费。所以这个额外的线程可以看作备用线程。

对于 Jetty 来说，也可以做这样的尝试，如果不能达到预期，可以逐步调整。



4.2.4 两个模型的对比及异步模型适用场景

从两个模型的对比来看，各有优劣，关键看如何选择，笔者再次对这两者进行对比，如表 4-5 所示。

表 4-5

| | 同步模型 | 异步模型 |
|-----------|--------------|---------------|
| 线程数量 | 大部分情况下比较多 | 较少 |
| I/O阻塞 | 有 | 无 |
| CPU调度开销 | 多 | 少 |
| 开发维护成本 | 低 | 高 |
| 需要很多线程的场景 | 达不到系统理论最高QPS | 比较接近系统理论最高QPS |

那么在什么场景下使用异步模型比较合适呢？举两个例子。

比如 DNS 服务，又比如广告服务，尤其是在视频网站上打开一个视频的时候，先来一段广告，这个广告服务使用异步模型比较合适，广告放完之后视频网站会告诉用户的浏览器去哪里下载视频（GSLB），这个视频地址路由服务使用异步模型也比较合适，因为这两种服务一个选择视频地址，另一个选择要播放的广告，具备下述特点。

- CPU Time 都比较小。
- 后面依赖的数据是可以异步获得的。
- 该服务在同步模型下 Wait Time 比较大（线程需要阻塞在网络 I/O 上读写广域网的请求）。
- 在同步模型下 QPS 与理论值差距太大。

在满足这些条件的场景下使用异步模型效果都会比较好。因为如果在这个场景下使用同步模型，那么 Wait Time 确实会比较大，而 CPU Time 本身比较小，需要大量线程才能达到预期的 QPS，大量线程带来的影响前面也讲过了，最终会降低系统的最大 QPS。

如果应用本身 CPU Time 比较大，通过计算或者压力测试，几十个或者四五百个线程就达到了系统预期的 QPS，尤其是搜索 Web 应用这种场景使用异步模型就不那么合适了。

总之，不管是同步模型还是异步模型都是满足最大 QPS 公式的，只是因为使用了非阻塞式 I/O，降低了整体的 Wait Time，导致所需的业务线程数降低了。异步模型虽然在线程数和 QPS 方面有优势，但是如上所述它也有劣势，我们要根据应用的特点来判断使用同步模型还是异步



模型，顺势而为。

4.2.5 小结

前面从公式推导、数据论证、权威资料佐证这三个角度证明了如图 4-10 所示的公式在特定条件下，从宏观上来讲是存在的。

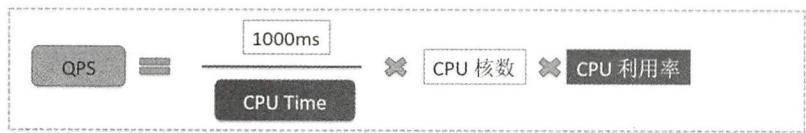


图 4-10

从该公式中，可以延伸推导出下面几点：

- (1) QPS 是由 CPU Time 和 CPU 利用率及 CPU 核数决定的。
- (2) 最佳线程数是由 CPU Time 和 Wait Time 及 CPU 核数、CPU 利用率决定的。
- (3) 多线程下 CPU Time 增加，以及 Admahl 定律中影响加速比的因素。
- (4) CPU Time 是由数据结构和算法决定的。
- (5) CPU 利用率与架构和串行、编程模型和系统中有无其他瓶颈相关。
- (6) 要做性能优化，必须考察 CPU Time 降低的百分比和 CPU 利用率提高的百分比。
- (7) 性能优化同时要考虑串行和并行的比例。
- (8) 处理某个业务的最佳线程数存在一个临界点，超过这个临界点的线程数，QPS 会有所下降，RT 也会明显增加。
- (9) 选择同步模型还是异步模型，要根据应用的特征仔细斟酌。

关键是通过最大 QPS 公式和实际测量数据的协同计算证明：显然在某电商的活动页面上 Gzip 是影响 CPU Time 的最大因素，排名第二的是 PHP 代码消耗的 CPU Time，而通过预压缩之后，Gzip 的开销和 PHP 代码的开销都被节省了，所以出现了 QPS 被提升很多倍的情况。



4.3 数据结构对性能的影响

对任何系统而言，性能都是整个系统架构中重要的一环，所以研究提高性能的手段成为了程序员的必修课之一，对于架构师来说更是重中之重。研究如何提高性能需要了解很多知识，包括数据结构、常用算法、网络、线程等，这些都需要不断地学习和实践才能掌握。也只有这样，我们写出来的代码才能让机器更好地执行，产生更好的执行效果。

不过代码除交给机器执行外，也必须能够让其他人轻松地维护，如果某个程序员写的代码在别人那里难以理解、难以扩展，那么这段代码的成本是很高的。所以一个好的程序员既要能写出容易被机器理解的代码（执行效率高），又要能写出让其他程序员容易理解的代码（维护成本低）。

程序的执行效率和可维护性就像天平的两端，有时候我们偏重执行效率，有时候我们偏重可维护性，更多的时候，我们力求选择两者的一个平衡点（或者可以“兼得”的方案）。能够根据程序的运行场景正确地选择天平状态的人，才是高手。在一个需要兼顾效率和可维护性的场景下一味地追求效率有可能只是为了炫耀自己的技术，反而增加了他人维护的成本。而在一个需要极高性能的场景下大谈特谈面向对象、设计模式之类的话题，也不是那么合适。所以一个好的程序员不但要学习数据结构和算法，也要学习面向对象编程和设计模式等，更进一步要学会在不同场景下使用不同的技能。

显然，笔者不会谈如何设计高可维护性和高可扩展性架构，本节将通过几个案例来谈谈数据结构对性能的影响。

4.3.1 HashMap 的问题

HashMap 是 Java 编程中最重要的数据结构之一，几乎每一段程序都离不开它，程序离不开 HashMap 就好像汽车离不开车轮一样。

在任何一个软件公司里人员构成都是梯队型的，总有在代码评审时遗漏的代码或者其他原因导致的问题代码。下面来看一个例子：

```
Map<Integer, Integer> map = new HashMap<Integer, Integer>();
for (int k = 0; k < 100000; k++) {
    map.put(k, k)
}
```




```
for(String key : map.keySet()) {  
    String value = map.get(key);  
    //执行其他逻辑  
}
```

这个例子看着非常简单，第一步将 10 万条数据放到 HashMap 中；第二步，取出来，好了。这只是给人的第一印象，实际上这段代码有两类问题。

1. resize 的问题

创建 HashMap 时没有根据已知的元素数量进行初始化，导致在对数据执行 Put 的时候需要执行 resize 很多次，在执行 resize 前需要创建新数组，并重新执行元素的 Hash。老的数组又要被回收，对 CPU 和垃圾收集都非常不友好，在连续插入大量数据时会严重降低性能。如果只是在程序初始化时执行一次，那么问题也不会特别大；如果每次请求过来时都有类似的逻辑，那么还是会带来不小的影响。具体的影响有多大，取决于该段代码的 Thread CPU Time 占 CPU Time 的比例。

2. 多余的 Hash 运算的问题

在循环取元素的时候，通过 keySet，而没有使用 EntrySet，导致在重复 map.get() 时出现多余的 Hash 运算，这些运算是在浪费 CPU 的时钟周期。

如果对 HashMap 的构造不甚了解，那么很容易写出上面的代码，而且现实中这样的代码不在少数。接下来详细地看一下 HashMap 的结构，以帮助我们解决上述问题，同时在今后的工作中更好地使用 HashMap。

4.3.2 HashMap 的结构

要知道 HashMap 是什么，首先要搞清楚它的数据结构。在 Java 编程语言中，最基本的结构有两种（其他编程语言也类似）：

- 数组。
- 模拟指针（引用）。

基本上所有的数据结构都可以用这两个基本结构来构造，HashMap 也不例外。HashMap 实际上是一个数组和链表的结合体（在数据结构中，一般称之为“链表散列”），请看图 4-11，纵向表示数组，横向表示数组元素（实际上是一个链表）。

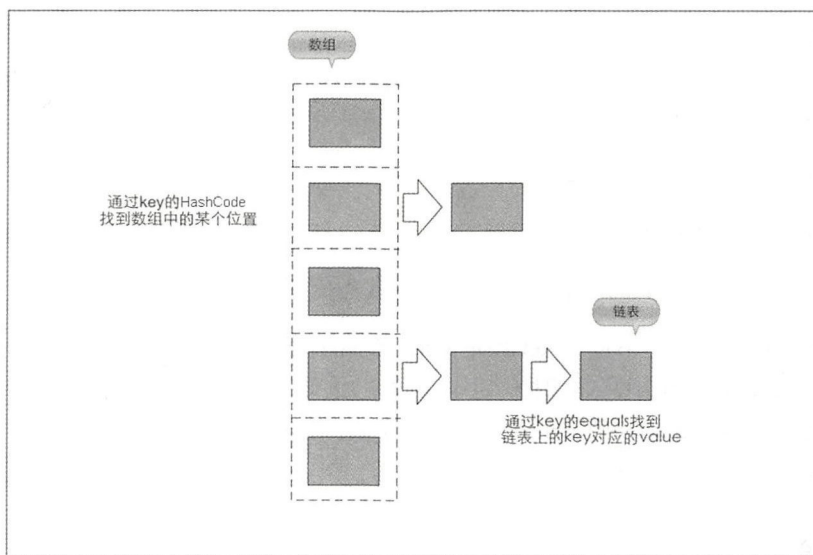


图 4-11

从图 4-11 中可以看到，一个 HashMap 就是一个数组结构，当新建一个 HashMap 的时候，就会初始化一个数组。来看看 Java 代码：

```
transient Entry[] table;

static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;
    .....
}
```

上面的 Entry 就是数组中的元素，它持有指向下一个元素的引用，这就构成了链表。所以，HashMap 中的主要结构就是“数组+链表”。

接下来，数组中的每一个元素，笔者都会用桶（bucket）来描述。

那么使用 HashMap 的时候是如何操作这个数组和其中的链表的呢？

1. 往 HashMap 中 put 元素时

（1）根据 key 的 Hash 值，通过 Hash 算法计算得到这个元素在数组中的位置（即下标），



也就是桶，然后就可以把这个元素放到对应的桶中了。

(2) 如果这个桶里已经存放其他元素了，那么在这个桶里的元素将以链表的形式存放，新加入的元素放在链头，最先加入的元素放在链尾。

2. 从 HashMap 中 get 元素时

(1) 计算 key 的 hashCode，通过 Hash 算法找到数组中对应位置的那个桶。

(2) 通过 key 的 equals 方法在桶里的链表中找到需要的元素。

从上述流程可以看到，不管是 get 还是 put，都离不开一个关键字，就是 Hash。通过 Hash 算法计算某个 key 在数组中具体的桶，可以想象，如果每个桶里的链表只有一个元素，那么 HashMap 的 get 效率将是最高的，但是理想总是美好的，现实中总有困难需要我们去克服。这个困难就是碰撞。

4.3.3 碰撞

什么是碰撞 (collision)？拿 HashMap 举例，如果两个 key 通过 Hash 算法计算之后在数组中得到的位置相同，即在同一个桶里，那么就是碰撞了。

如果在一个有多个元素的 HashMap 中，通过 Hash 算法运算之后有 6 个元素都在同一个桶中，那么会出现如图 4-12 所示的恐怖状况。

当执行 get 操作时，首先根据 Hash 算法算出桶的位置，然后需要在这个桶里执行 6 次 equals 方法才能找到对应的元素。

这意味着桶里的数据越多，碰撞得越厉害，get 时间越长；桶里的数据越少，get 性能越高。尤其在没有碰撞的时候，每个桶里只有一个元素，那么整个 Map 的性能是最优的。

所以，我们要做的就是努力减少碰撞，减少碰撞有如下两种方法。

(1) 优化 Hash 算法。

(2) 增加数组的大小。

接下来笔者详细讲讲这两种方法。

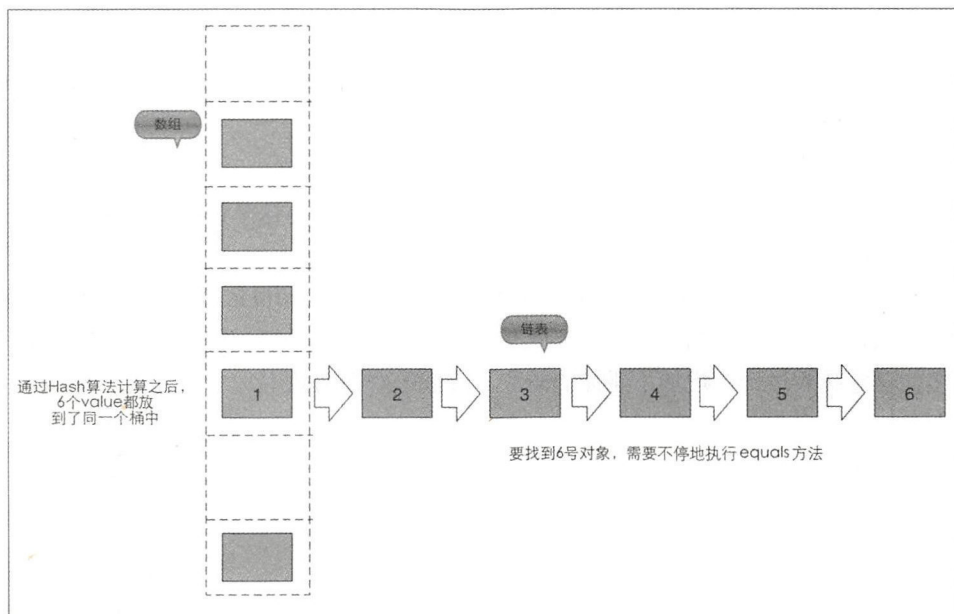


图 4-12

4.3.4 Hash 算法

1. Hash 算法的目标

Hash 算法的目标是：既要性能高，又要碰撞少。

2. 计算 key 在数组中的具体位置

```
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

首先计算 key 的 Hash 值（这个 HashCode 并不是 hashCode 方法返回的，而是 HashMap 中 Hash 方法返回的 Hash 值），然后跟“数组的长度-1”做一次“与”（&）运算。比如数组的长度是 2^4 ，那么 HashCode 就会和“ 2^4-1 ”做“与”运算。很多人都有这个疑问：为什么 HashMap 的数组初始化大小都是 2 的指数幂时，HashMap 的效率最高？

很简单，因为 2 的指数幂减 1 的二进制除符号位外全是 1。那么问题又转化成：为什么和二进制全是 1（符号位除外）的数进行“与”运算效率最高呢？



1) $h \& (\text{length}-1)$ 原理

如图 4-13 所示，上边两个数组长度为 1024 (2^{10})，下边两个数组长度为 1023，两组的 HashCode 相同。但是很明显，当它们和 $1023-1(1111111110)$ 进行“与”运算的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上，这就产生了碰撞。两个 HashCode 会被放到同一个链表的同一位置上，那么查询的时候就需要遍历这个链表，这样就降低了查询效率。同时，我们也可以发现，当数组长度为 15 时，HashCode 的值会与 14 (1110) 进行“与”运算，最后一位永远是 0，而 0001、0011、0101、1001、1011、0111、1101 这几个位置永远都不能存放元素，空间浪费相当大，更糟糕的是，在这种情况下数组可以使用的位置比数组长度少了很多，这意味着进一步增加了碰撞的概率，降低了查询的效率！



图 4-13

所以，当数组长度为 2 的 n 次幂的时候，再加上 $h \& (\text{length} - 1)$ 这个方法，不同的 key 计算得到 index 相同的概率较小，数据在数组上分布比较均匀，碰撞的概率小，查询时不用遍历某个位置上的链表，这样查询效率比较高。

2) HashMap 中数组的大小

说到这里，再回头看一下 HashMap 中默认的数组大小，查看源代码可以得知是 16，为什么是 16 而不是 15，也不是 20 呢？因为 16 是 2 的整数次幂，在数据量小的情况下 16 比 15 和 20 更能减少碰撞，进而提高查询的效率。





所以，在存储大容量数据的时候，最好预先指定 `HashMap` 的大小为 2 的整数次幂。就算不指定，也要以大于且最接近指定值大小的 2 次幂来初始化，代码如下（在 `HashMap` 的构造方法中）。

```
int capacity = 1;
while (capacity < initialCapacity)
    capacity <<= 1;
this.loadFactor = loadFactor;
threshold = (int)(capacity * loadFactor);
table = new Entry[capacity];
init();
```

取 Hash 值

在 JDK 中，取 Hash 值并不是简单地拿到 `HashCode` 就完事了，还有几个位运算。

```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

如果需要被 Hash 的 key 是字符串，则会调用 `stringHash32` 方法进行 Hash 后返回，如果 key 不是字符串，则进行一系列位移后返回最终结果。

多年前，笔者在看 JDK 的 `HashMap` 源代码时，字符串的 Hash 是没有 `sun.misc.Hashing.stringHash32((String) k)` 这一句的，现在有了。非字符串对象的 Hash 对于非字符串对象拿到 `HashCode` 之后，还要执行 `h ^= (h >>> 20) ^ (h >>> 12)` 和 `h ^ (h >>> 7) ^ (h >>> 4)` 两行代码，这两行代码的作用是什么？请看下面的代码示例。

```
public static void hash (int h) {
    int h1 = h >>> 20;
    int h2 = h >>> 12;
    int h3 = h1 ^ h2;
    int h4 = h ^ h3;
    int h5 = h4 >>> 7;
```





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

```

int h6 = h4 >>> 4;
int h7 = h5 ^ h6;
int h8 = h4 ^ h7;

printBin ( h );
printBin ( h1 );
printBin ( h2 );
printBin ( h3 );
printBin ( h4 );
printBin ( h5 );
printBin ( h6 );
printBin ( h7 );
printBin ( h8 );
}

static void printBin ( int h ) {
    System.out.println ( String.format ( "%32s",
        Integer.toBinaryString ( h ) ).replace ( ' ', '0' ) );
}

```

当输入参数为 `int h = 0xffffffff` 的时候，输出结果为：

```

111111111111111111111111111111111111 //原始值
000000000000000000000001111111111111 //int h1 = h >>> 20;
000000000000011111111111111111111111 //int h2 = h >>> 12;
000000000000011111111000000000000000 //int h3 = h1 ^ h2;
111111111111000000000111111111111111 //int h4 = h ^ h3;
000000011111111111100000000111111111 //int h5 = h4 >>> 7;
000011111111111110000000011111111111 //int h6 = h4 >>> 4;
0000111000000000011100000111000000 //int h7 = h5 ^ h6;
1111000111110000011101111000111111 //int h8 = h4 ^ h7;

```

从上述代码的输出来看，原来 `111111111111111111111111111111111111` 经过一系列的位运算之后，得到的 `HashCode` 是 0 和 1 较为均匀的一个组合。那么为什么不直接返回原值，而要大费周章地拿到一个位移之后 0 和 1 较为均匀的组合呢？

也很简单，我们来看例子。假设数组大小还是 1024，有两个 key，各自的 `HashCode` 分别为 1025 和 2049。如果不经 Hash 方法，直接拿 `HashCode` 进行 `indexOf` 运算，那么结果如图 4-14 所示。



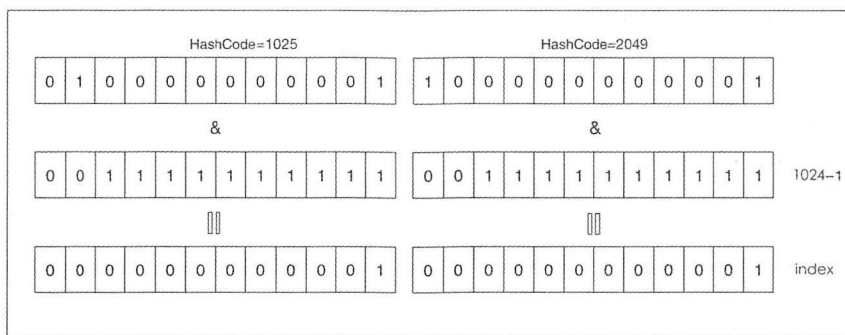


图 4-14

如果使用 Hash 方法之后进行 indexFor 运算，结果如图 4-15 所示。

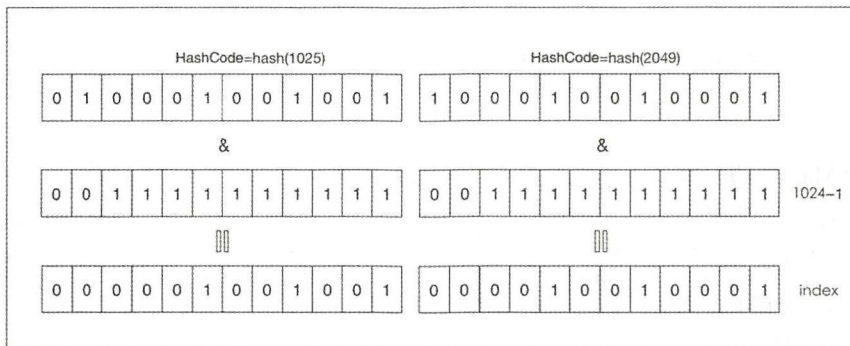


图 4-15

从图 4-15 可以看出，经过较为均匀的混合之后，1025 和 2049 对应的 key 计算出的 index 不再相同，这意味着冲突被解决了。这便是 HashMap 中位移运算的作用。

如果没有均匀分布，那么 1025 和 2049 在高位是一样的，而和 1024-1 做“与”运算时，低位相同，从而产生碰撞。

如果做了均匀分布，那么在低位上 1025 和 2049 执行 Hash 运算后的结果就不一样了，即减小了碰撞的概率。

总结一下 HashMap 中 Hash 算法减少碰撞的两个做法：

- Hash 方法中存在奇怪的位移运算，以使最终的 Hash 值中 0 和 1 较为均匀。
- 使用 $h \& (\text{length} - 1)$ 来进行桶的定位，length 应为 2 的指数幂。



为什么不用取模？用取模当然也是可以的，尤其是在早期的某些数据结构中，比如 HashTable 中的 get 是这样的：

```
public synchronized V get(Object key) {
    Entry tab[] = table;
    int hash = hash(key);
    //取模来判断元素的 index
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return e.value;
        }
    }
    return null;
}
```

不过由于该方案的碰撞率和性能与 HashMap 中的方案相比皆低，已经很少用了。

1) resize

当 HashMap 中的元素越来越多的时候，碰撞的概率也就越来越高（因为数组的长度是固定的），所以为了提高查询效率，就要对 HashMap 的数组进行扩容。数组扩容操作也会出现在 ArrayList 中，所以这是一个通用的操作。很多人对它的性能表示怀疑，不过想想“均摊”原理，就释然了。在 HashMap 数组扩容之后，最消耗性能的点出现了：对原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是 resize。

HashMap 什么时候进行扩容呢？当 HashMap 中的元素个数超过数组大小 loadFactor 时，就会进行数组扩容，loadFactor 的默认值为 0.75。也就是说，在默认情况下，数组大小为 16，当 HashMap 中元素个数超过 $16 \times 0.75 = 12$ 的时候，就把数组的大小扩展为 $2 \times 16 = 32$ ，即扩大一倍，然后重新计算每个元素在数组中的位置。

这是一个非常消耗性能的操作，所以如果能预知 HashMap 中元素的个数，那么提前设置好元素的个数能够有效地提高 HashMap 的性能。比如，有 1000 个元素，就 new HashMap(1000)，但是理论上 new HashMap(1024)更合适，不过上面已经说过，即使是 1000 也会自动将其设置为 1024。

但是在这个例子中 new HashMap(1024)还不是最合适的，因为 $0.75 \times 1000 < 1000$ ，也就是说，为了让 $0.75 \times \text{size} > 1000$ ，我们必须 new HashMap(2048)才最合适，既考虑了“&”的问题，也避免了 resize 的问题。





2) key 的 hashCode 与 equals 方法改写

在 HashMap 的数据结构中讲了 get 方法的过程：首先计算 key 的 hashCode，找到数组中对应位置的某一元素，然后通过 key 的 equals 方法在对应位置的链表中找到需要的元素。所以，hashCode 与 equals 方法对于找到对应元素来说是两个关键点。

HashMap 的 key 可以是任何类型的对象，例如 User，为了保证两个具有相同属性的 User 的 hashCode 相同，需要改写 hashCode 方法，比如把 hashCode 值的计算与 User 对象的 ID 关联起来，只要 User 对象拥有相同 ID，那么它们的 hashCode 也能保持一致，这样就可以找到其在 HashMap 数组中的位置。如果这个位置上有多个元素，还需要用 key 的 equals 方法在对应位置的链表中找到需要的元素，所以只改写 hashCode 方法是不够的，equals 方法也是需要改写的。当然，按正常思维逻辑，equals 方法一般都会根据实际的业务内容来定义，例如，根据 User 对象的 ID 来判断两个 User 是否相等。

在改写 equals 方法的时候，需要满足以下三点。

- 自反性：a.equals(a)必须为 true。
- 对称性：如果 a.equals(b)=true，b.equals(a)也必须为 true。
- 传递性：如果 a.equals(b)=true，并且 b.equals(c)=true，a.equals(c)也必须为 true。

通过改写 key 对象的 hashCode 和 equals 方法，可以将任意的业务对象作为 Map 的 key（前提是确实有这样的需要）。

3) 性能测试

根据我们对 HashMap 结构的理解，可以看出之前的那个使用 HashMap 的代码片段中的两个明显的问题，第一个问题是 resize 的问题，带来大量的数组 copy 和 rehash，第二个问题是根据 keySet 中的 key 来遍历 HashMap 带来重复的 Hash 运算。可以把修改之后的代码和之前的代码做一个测试比较。

4) HashMap 的不合理使用

(1) 在特定情形下，没有合理地设置初始值

这涉及 resize，对 GC 和 CPU 都不友好，严重的时候会引起系统异常。

(2) 使用 key 时没有逻辑判断的依据

代码如下。





```
for(String element : list) {
    if(map.containsKey(element)) {
        String value = map.get(element);
        //执行其他业务逻辑
    }
}
```

这段代码中有明显的重复 Hash，因为 value 存在时，contains 和 get 执行了两次 Hash 运算，我们能简单地改成如下代码吗？

```
for(String element : list) {
    String value = map.get(element);
    if(value != null) {
        //执行其他业务逻辑
    }
}
```

理论上不行，因为两者的含义不一样，第一段代码判断 key 是否存在，第二段代码判断 value 是否为 null，如果遇到 key 存在、value 为 null 的场景，那么两段代码就不等价了。所以尽量用 value 是否存在来做逻辑判断，而不使用 key 是否存在来做逻辑判断。

也许在大部分场景下问题不大，但是在某些场景里，比如当 Hash 占用大量 CPU Time 时，这种优化会起到立竿见影的效果，笔者所在的团队就出现过这样的情况。

刚刚讲到了 HashMap 的正确使用和滥用的一些方式，但是还没有讲其错误的使用方式。我们知道 HashMap 不支持多线程操作，正确的做法是不要让多线程并发操作 HashMap，一旦这么做可能出现以下场景。

- 数据丢失（想想多线程操作链表或者数组会发生什么）。
- 死循环（再想想多线程操作数组或者链表会发生什么）。
- 抛出并发异常。

一个数据结构，正确的用法都是类似的（基于对这个数据结构的理解），而错误的用法却多种多样。所以为了避免出现“莫名其妙”的问题，应该尽可能地深入理解常用的数据结构。

4.3.5 题外话：ConcurrentHashMap 中的 Hash

上面讲到 HashMap 中 Hash 方法的精要及 indexOf 和 2 的指数幂之间的关系，接下来看看





ConcurrentHashMap 中的 Hash 和 HashMap 中的 Hash 有什么不一样。

```
private int hash(Object k) {  
    int h = hashSeed;  
  
    if ((0 != h) && (k instanceof String)) {  
        return sun.misc.Hashing.stringHash32((String) k);  
    }  
  
    h ^= k.hashCode();  
    h += (h << 15) ^ 0xffffcd7d;  
    h ^= (h >>> 10);  
    h += (h << 3);  
    h ^= (h >>> 6);  
    h += (h << 2) + (h << 14);  
    return h ^ (h >>> 16);  
}
```

ConcurrentHashMap 中使用的是 single-word wang/jenkins hash 算法的变种, 这个算法的优点是什么? 我们来看一下对某个 hashCode 进行运算之后的结论: 和 HashMap 的 Hash 方法相比, single-word wang/jenkins 算出的结果更加均匀了, 即连高位也均匀了。这样做的目的是什么呢?

其实要结合 ConcurrentHashMap 的 segmentForHash 方法来看, 在 get 方法中, 有这样的代码, 主要功能是选择 segment。

```
private Segment<K,V> segmentForHash(int h) {  
    long u = ((h >>> segmentShift) & segmentMask) << SSHIFT) + SBASE;  
    return (Segment<K,V>) UNSAFE.getObjectVolatile(segments, u);  
}  
Sun.Misc.Hashing.Stringhash32
```

对于 ConcurrentHashMap 的 Hash 方法来说, 不能直接使用 HashMap 的 Hash 方法, 否则碰撞增加、性能下降、CPU Time 增加, 最终将导致系统总的 QPS 下降。

不管是哪个 Hash 方法, 在新版的 JDK 中, 都对 String 做了特殊的 Hash 算法, 以取得更好的效果, 具体原理可以参考相关资料。

4.3.6 HashMap 综述

关于 HashMap 我们主要描述了以下几点。





- HashMap 的结构是由数组和链表实现的，其中 Hash 函数的实现有助于减少碰撞。
- 描述了 HashMap 中 resize 带来性能消耗的根本原因。
- 将普通的域模型对象作为 key 的基本要求。
- 不合理地使用数据结构会导致 CPU Time 增加，从而影响总的 QPS。
 - 死循环增加 CPU Time。
 - resize 增加 CPU Time。
 - 不好的 Hash 函数带来的碰撞会增加 CPU Time。
 - 重复的 Hash 运算会增加 CPU Time。

在精确地理解了这些内容之后，可以说：我们终于对 HashMap 略知一二了。

同时，如果读者也遇到类似的场景（有很多 bucket / slot，要根据 UserId 选择 bucket / slot），如何设计 Hash 算法才能够使所有的 UserId 均匀地落到不同的 bucket /slot 里？读者可以参考 HashMap 或者 ConcurrentHashMap 的实现。

4.3.7 均摊

1. 均摊分析原理

均摊分析是对一组操作的分析。特别是均摊分析允许处理这样一种情况， n 个操作在最差情况下的代价小于任何一个操作在最差情况下代价的 n 倍。均摊分析并不是把注意力集中到每个操作的单独代价上，再把它们加起来，而是看一组操作的整体代价，再把整体的代价分摊到每一个单独的操作上。

所以均摊的存在有其合理性，基本上在任何编程语言的 SDK 中都存在大量的符合均摊描述的数据结构，接下来介绍一下 Java 中的均摊。

2. 均摊在 Java 数据结构中的体现

数据结构基本都由两部分内容组成。

1) ArrayList

```
private void grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
}
```





```
        elementData = Arrays.copyOf(elementData, newCapacity);  
    }
```

2) StringBuilder(StringBuffer)

```
void expandCapacity(int minimumCapacity) {  
    int newCapacity = value.length * 2 + 2;  
    if (newCapacity - minimumCapacity < 0) {  
        newCapacity = minimumCapacity;  
    }  
    if (newCapacity < 0) {  
        if (minimumCapacity < 0) // overflow  
            throw new OutOfMemoryError();  
        newCapacity = Integer.MAX_VALUE;  
    }  
    value = Arrays.copyOf(value, newCapacity);  
}
```

3) ByteArrayOutputStream

```
private void grow(int minCapacity) {  
    int oldCapacity = buf.length;  
    int newCapacity = oldCapacity << 1;  
    if (newCapacity - minCapacity < 0)  
        newCapacity = minCapacity;  
    if (newCapacity < 0) {  
        if (minCapacity < 0) // overflow  
            throw new OutOfMemoryError();  
        newCapacity = Integer.MAX_VALUE;  
    }  
    buf = Arrays.copyOf(buf, newCapacity);  
}
```

3. 均摊案例

均摊原本的理念是为了说明把整体的代价分摊到每一个单独的操作上是可行的，而且确实在绝大多数场景中都是合理有效的，但是在实际使用的过程中，在少数场景下还是会出现误用，导致 CPU Time 增加，Wait Time 有时也会增加，这不但降低了系统的 QPS，还降低了系统的 RT。接下来，我们看一个案例。

JackRabbit 是一个 content repository，即内容管理系统，有点类似云盘，它维护着云盘中的目录数据和真实的文件数据，对于 JackRabbit 来说，目录数据（文件路径、文件名之类）和文件本身都是可以检索的。





我们分析一下，如表 4-6 所示。

表 4-6

| 检索内容 | 备注 |
|----------|---|
| 文件名和文件路径 | 这类属性称为元数据，主要是一个树形结构，“叶节点”是文件信息描述，“其他节点”是目录信息描述 |
| 文件内容 | 由于文件的格式是多种多样的，比如PDF、Word、Excel等，所以要对这些文件做文本检索，首先要做文本提取，而不同的文件内在的组织格式不一样，所以对每种单独的文件格式都要做抽取 |

虽然 Apache 上有抽取文件内容的工具包，但以前这些包还不算稳定，经常会出现一些莫名的问题，而搞清楚每种文件的格式又是一个旷日持久且 ROI 很低的投入。

时隔不久，该问题再次出现，虽然出现在不同的场景下，但现象有雷同之处。这次更离谱：在某个他人的项目中，Eden 区还有 300MB 空间的时候发生了 OOM（Out Of Memory，内存溢出）。

不出意外的话，应该也是大的 char[]或者 byte[]达到极限所带来的问题。思考一下，我们常见的 char[]的应用有哪些？无非就是 StringBuilder、StringBuffer。

```
char value[];
```

数组是无法扩容的，当数组空间不够的时候，会创建一个更大的数组，把原来数组的数据全部复制到新的数组中，再把新的数据追加到新的数组的后面。这是典型的均摊的实现。

下面我们来详细考量一下该理论在 StringBuilder 类的空间复杂度和时间复杂度。假设我们有 100 个 char 需要放到一个 StringBuilder 中，根据它的实现，一共会有 3 次扩容，并且有 100 次 append 操作，假设一次扩容（建更大的数组，然后执行复制）的时间消耗是 m ，append 操作的时间消耗是 n ，那么在这次 StringBuilder 的使用过程中，总的时间消耗是：

$$\text{cost} = (3m + 100n) / 100$$

而且 $m \gg n$ 。我们在使用 StringBuilder 的过程中，要思考的是如何降低 cost 的值。当然降低 $3m$ 是最好的，但是对于 m 我们无能为力，那么就对“3”下手吧，如果已知 char 的总数，我们就可以把 3 降下来，只要通过 `StringBuilder sb = new StringBuilder(100)`，就可以避免 3 次扩容，这样 $\text{cost} = (100n) / 100 = n$ 。

这个时间问题是我们避免 3 次扩容的最佳理由吗？不是！为什么？因为看上去 $(3m + 100n) / 100$ 并不比 n 高到哪里去。既然时间复杂度还不是最佳理由，我们的目光自然而然地转到空间复杂度上。



其实均摊定义对于 `StringBuilder` 之类的实现来说只解释了时间复杂度的问题，并没有涉及空间复杂度。由于我们并不能直接操作内存的分配，所以在 JVM 中问题显得更加神秘。

第一次扩容时，`char[16]` 不够，重新创建了一个 `char[(16+1) × 2]`，第二次扩容时重新创建了 `char[(34 + 1) × 2]`，第三次扩容时重新创建了 `char[(70 + 1) × 2]`。这个时候一共产生了 4 个数组对象：`char[16]`、`char[34]`、`char[70]`、`char[142]`。其实只是“append”了 100 个 `char` 而已，消耗的空间最大却有可能达到 262 个 `char`。

262 个 `char` 还不至于消耗很大空间，因为我们只做了 100 次 `append`，如果我们有 1 000 000 个 `char` 呢？我们来算一下时间消耗和空间消耗。总的扩容次数为 16（在 1 000 000 个 `char` 的情况下）的时间消耗：

$$(16m + 1\,000\,000n) / 1\,000\,000$$

空间消耗：

$$34 + 70 + \cdots + 294910 + 589822 + 1\,179\,646 = 2\,359\,244$$

在整个操作的生命周期内总共会消耗 2 359 244 个 `char` 的空间，但是由于 GC 的作用，同一时刻最大的消耗至少为 $589\,822 + 1\,179\,646 = 1\,769\,468$ 。空间消耗至少翻倍（为什么是至少？因为前面创建的 `char[]` 如果还没有被回收，那么消耗的空间就会更大，最大会达到 2 359 244 个 `char`。如果是多线程在做这样的事情，那么消耗的空间数还要乘以线程数。比如原来一个线程这样的操作只浪费 2MB 空间，但是 100 个线程其实就浪费了 200MB 空间）。

再深入考虑，难道这样做只是增加了空间的消耗吗？绝不是。我们还可以从 JVM 的角度来考虑一下这个问题。前面 15 个 `char[]` 的创建对于 JVM 的 `young` 区的复制算法来说也是一个不必要的负担，因为如果有 100 个线程在做这个任务，同一时间可能产生 $15 \times 100 = 1500$ 个多余的对象。在 GC 的时候，这些对象需要执行 `mark`、`copy`（前面临时的 `char[]` 至少要执行 `copy` 一次，也就是说它们必须进入一次 `From Space` 或者 `To Space`），其实在一些场景下，这种操作完全可以避免，方法就是给 `StringBuilder` 赋予一个合理的值。

4. 小结

在用 `StringBuilder` 之类的方法实现扩容的时候，需要考虑均摊的相关影响，如：

- 时间消耗。
- 空间消耗。
- 对 JVM 的影响。



通过前面的分析可以让我们更好地使用 Java SDK 中类似的实现类，写出性能更高、更健壮的代码。

4.4 算法设计不合理带来的性能问题

在数年前的一次大促中，按照以往的经验，下半年是业务发展的高峰时期，所以技术保障工作的要求会比上半年的大促更高，而在技术保障中，压力测试又是重要的一环。本来根据安排，笔者专注在平台升级的工作上，但是由于紧急情况，调入了压力测试小组，也是因为调入了压力测试小组，所以和大家一起发现了很多问题，也解决了很多问题。其中有一个问题就是分布式缓存系统中关于 mget 的一个漏洞，接下来我们从现象说起，看看这个问题是怎么回事。

4.4.1 某应用 A 的现象

在对某应用进行压力测试时，笔者发现在 A Server 中 Load 较高的一台机器上，CPU 中有一个核占用率非常高，于是我们立刻根据 PID 算出线程号（使用 top -H 调用出来的 PID 是十进制数值），如图 4-16 所示。

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|-------|----|----|-------|------|-----|---|------|------|-----------|---------|
| 9489 | admin | 15 | 0 | 5132m | 4.2g | 29m | S | 49.2 | 56.8 | 165:30.56 | ava |
| 29178 | admin | 15 | 0 | 5132m | 4.2g | 29m | S | 10.5 | 56.8 | 0:02.80 | ava |
| 17730 | admin | 15 | 0 | 5132m | 4.2g | 29m | R | 9.8 | 56.8 | 0:04.18 | ava |
| 29159 | admin | 16 | 0 | 5132m | 4.2g | 29m | S | 9.8 | 56.8 | 0:00.99 | ava |
| 29216 | admin | 15 | 0 | 5132m | 4.2g | 29m | S | 7.5 | 56.8 | 0:02.34 | ava |
| 29218 | admin | 16 | 0 | 5132m | 4.2g | 29m | R | 6.2 | 56.8 | 0:01.94 | ava |
| 17756 | admin | 15 | 0 | 5132m | 4.2g | 29m | S | 5.6 | 56.8 | 0:01.39 | ava |
| 17748 | admin | 15 | 0 | 5132m | 4.2g | 29m | R | 5.2 | 56.8 | 0:03.14 | ava |

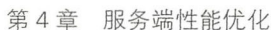
图 4-16

然后我们又根据线程号在 jstack 中找到线程名（jstack 中的线程号是十六进制，所以要把 top -H 中的十进制 PID 转换成十六进制）。

于是笔者就发现了一个线程，叫作 callbacktaskscan.scancallbackTasks。

4.4.2 某应用 B 的现象

同时笔者路过另外一个同事的座位时，该同事反馈他的 B 应用 CPU 消耗也很高，这个应用在 CPU 利用率达到 200% 以上时，有一个线程花的时间特别多，一直高居各线程榜首，如图 4-17 所示。

图 4-17

笔者把这个线程号转换成十六进制后，通过 jstack 搜索发现如图 4-18 所示的结果。

图 4-18

还是 xxxxxx.scanCallbackTasks。

两个不同的应用，A 应用是在压力测试时发现的，B 应用是在非压力测试时间发现的，所以看上去那个线程也只占用了 23% 的 CPU 资源。但是后面可以看到，机器的负载越高，这个线程消耗的 CPU 资源就越高。高峰时期这个线程会消耗掉一个核。

回想一下，在之前的大促中也遇到过这个问题，当时笔者立刻去翻看相关的代码，由于时间紧迫，没能彻底把代码翻看一遍，所以并没有找到原因。这次又遇到相同的问题，笔者想不能再错过，于是将其代码找出来进行了详细的阅读。

4.4.3 分析

经过了解，笔者发现 A 应用和 B 应用在使用分布式缓存的时候，都使用了 `mget` 方法，所以笔者把精力放到了这个方法上。



为了避免版本问题，将分布式缓存的客户端升级到最新版本，然后通过代码阅读，笔者总结了整个流程，如图 4-19 所示，有些细节没有包含在流程图里。

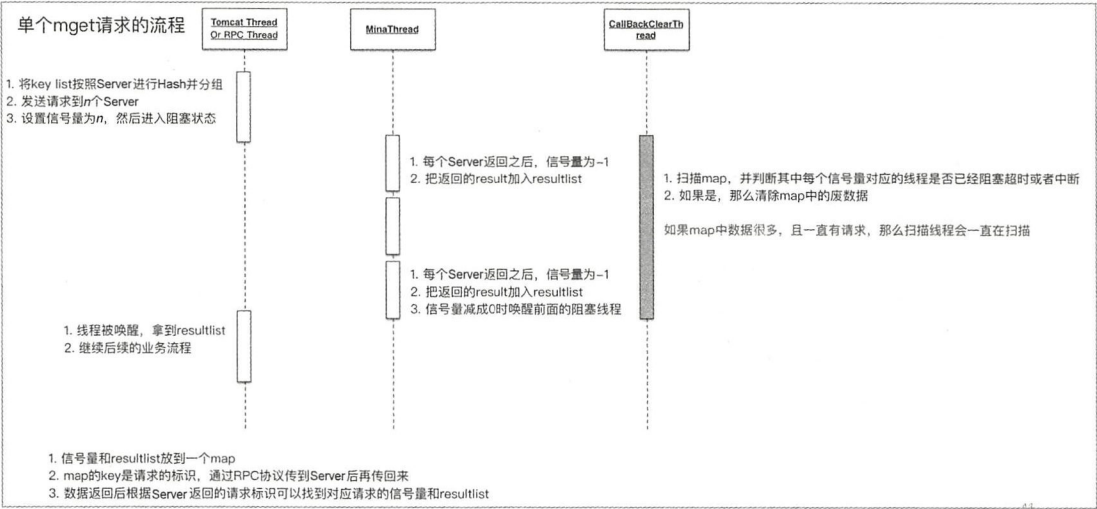


图 4-19

那么 CallBackTasksScan 线程如何清理 serverdataTasks 中已经返回的数据或者已经超时的任务呢？看如图 4-20 所示这段伪代码。

```
public void run() {  
    //这里是一个死循环，不断做返回结果的检查  
    while(isRunning){  
        /*这里最关键的代码是循环，只要数据没返回，或者超时时间没到，这个循环就不做其他事情，就是两个 if 判断连续不断地跑。*/  
        scanCBTasks(tasks);  
        /*此处处理其他逻辑，如果没有 request 在等待 response 返回，后续的代码会“sleep”一小会儿。*/  
    }  
}
```

图 4-20

只要数据没返回，或者超时时间没到，这个循环就不做其他事情，即两个 if 判断连续不断地跑。尤其是有很多 mget 请求且服务器端 RT 稍微长的时候（比如 5ms），这个情况更加明显。这个线程根本停不下来，一直以极高的速度大量消耗 CPU 资源，这样的“死”循环，就会导致一个核被 100% 占用。



4.4.4 方案

CPU 资源消耗过高的原因找到了，那么使用什么解决方案呢？由于之前从无到有带团队实现过整套 RPC，笔者在这方面有过经验（当然 RPC 里并没有这样的逻辑，RPC 的逻辑和分布式缓存中 get 的逻辑是相似的），总的来说思路是一样的，即谁知道、谁清理。

我们来看一下，清理的两种条件：

（1）数据从 socket 重返回，并且通知对应的线程之后（可能是 Tomcat 线程或者 RPC 线程）需要清理出 serverdataTasks。

（2）数据长时间不返回，超过了规定时间，需要清理出 serverdataTasks。

本着谁知道、谁清理的逻辑，我们看看在第一种条件下，谁知道数据返回，毫无疑问，是 mina 的 workthread 调用了 messagereceived 方法，它是知道的，所以应该在如图 4-21 所示的方法中清理。

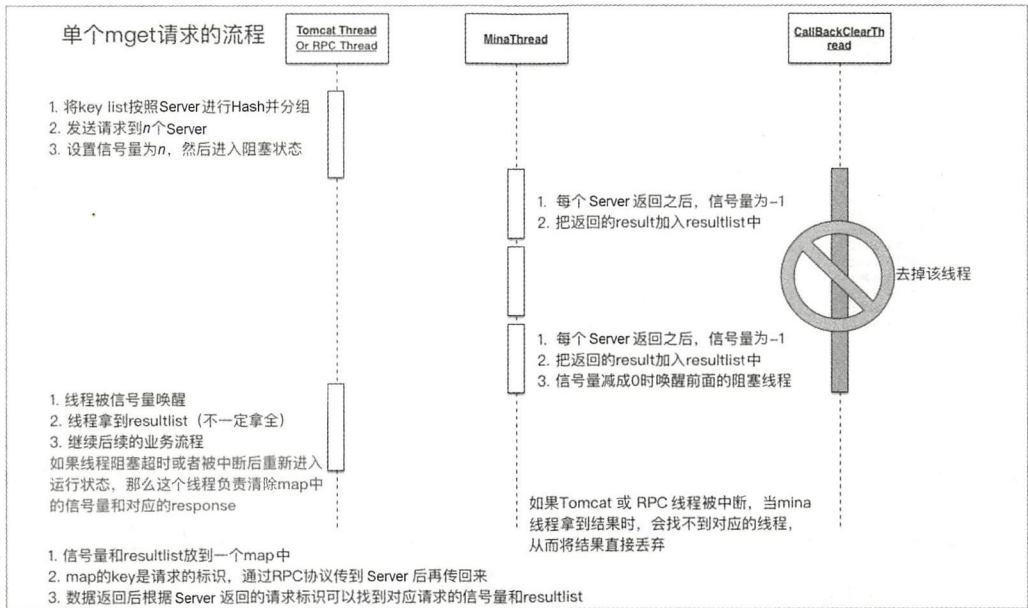


图 4-21

如此就不需要一个死循环来检查这个 map 中是否有对象需要被清理，正常返回和异常返回都已经由对应的线程处理了。这样一来，就不会有一个线程在请求高峰时期不断地扫描某个 map 而造成 CPU 中一个核的浪费。



4.4.5 验证

在预发布机器上，把修改之后的代码放上去，验证功能，功能没有问题，然后将这份代码发到一台线上的机器上，拿一台配置一模一样（物理机配置一样）的虚拟机，对整个集群进行压力测试，查看 CPU 的使用情况。

同一时间段，两台机器进行对比，01 是新代码的机器，02 是原始代码的机器，下面来看一下机器在压力测试时的表现。

图 4-22 是笔者改过代码的运行效果，可以看出 CPU 的 idle 是 65.2%，而且没有一台消耗 CPU 资源特别高的机器。

top - 00:05:03 up 151 days, 20:30, 4 users, load average: 3.65, 1.98, 1.10
Tasks: 759 total, 20 running, 739 sleeping, 0 stopped, 0 zombie
Cpu(s): 29.4%us, 3.8%sy, 0.0%ni, 65.2%id, 0.0%wa, 0.0%hi, 1.6%si, 0.1%st
Mem: 12853504k total, 10101308k used, 1952196k free, 361492k buffers
Swap: 2000116k total, 120k used, 2007996k free, 6462832k cached

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | MEM | TIME+ | COMMAND |
|-------|-------|----|----|-------|------|-----|---|------|------|---------|---------|
| 10931 | agent | 15 | 0 | 469m | 93m | 12m | S | 47.5 | 0.8 | 0:01.44 | java |
| 11460 | agent | 15 | 0 | 469m | 93m | 12m | S | 44.9 | 0.8 | 0:01.36 | java |
| 10769 | agent | 15 | 0 | 469m | 93m | 12m | S | 41.6 | 0.8 | 0:01.26 | java |
| 11094 | agent | 15 | 0 | 469m | 93m | 12m | S | 40.9 | 0.8 | 0:01.24 | java |
| 1217 | admin | 16 | 0 | 5170m | 2.3g | 28m | R | 2.6 | 20.1 | 0:13.36 | java |
| 1341 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 2.3 | 20.1 | 0:01.75 | java |
| 1354 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 2.3 | 20.1 | 0:02.52 | java |
| 1220 | admin | 15 | 0 | 5170m | 2.3g | 28m | R | 2.0 | 20.1 | 0:13.46 | java |
| 1366 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 2.0 | 20.1 | 0:01.79 | java |
| 1385 | admin | 15 | 0 | 5170m | 2.3g | 28m | R | 2.0 | 20.1 | 0:03.31 | java |
| 14624 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 2.0 | 20.1 | 0:00.06 | java |
| 1215 | admin | 16 | 0 | 5170m | 2.3g | 28m | R | 1.6 | 20.1 | 0:13.03 | java |
| 1219 | admin | 16 | 0 | 5170m | 2.3g | 28m | R | 1.6 | 20.1 | 0:06.43 | java |
| 1346 | admin | 16 | 0 | 5170m | 2.3g | 28m | S | 1.6 | 20.1 | 0:00.14 | java |
| 1397 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 1.6 | 20.1 | 0:03.56 | java |
| 1407 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 1.6 | 20.1 | 0:03.03 | java |
| 1417 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 1.6 | 20.1 | 0:59.50 | java |
| 13927 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 1.6 | 20.1 | 0:02.24 | java |
| 13953 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 1.6 | 20.1 | 0:02.39 | java |
| 1118 | admin | 16 | 0 | 5170m | 2.3g | 28m | R | 1.3 | 20.1 | 0:12.16 | java |
| 1358 | admin | 15 | 0 | 5170m | 2.3g | 28m | R | 1.3 | 20.1 | 0:01.38 | java |
| 1364 | admin | 16 | 0 | 5170m | 2.3g | 28m | S | 1.3 | 20.1 | 0:02.27 | java |
| 1370 | admin | 16 | 0 | 5170m | 2.3g | 28m | S | 1.3 | 20.1 | 0:00.43 | java |
| 1373 | admin | 15 | 0 | 5170m | 2.3g | 28m | S | 1.3 | 20.1 | 0:03.81 | java |

图 4-22

图 4-23 是没有改过的代码的运行效果，图中 CPU 的 idle 只有 48.6%，而且有一个线程特别消耗 CPU 资源，由于两个截图时间间隔是 5~10s，准确来讲，两者的机器情况相差了 10s。

那么图 4-23 中消耗 73.6%资源的线程是谁呢？我们来看一下，如图 4-24 所示。

还是这个 CallbackTaskScan，而第一台机器则没有这样的线程消耗这么高比例的 CPU 资源。

图 4-22 和图 4-23 是经过几分钟的长时间观察的情况，结果基本保持不变。

同一机器，在不同时间段进行两次压力测试后（第一次压力测试没有新代码，第二次压力测试上了新代码），笔者又到监控平台去查看这台测试机器在不同时间段的表现，如图 4-25 所示，Load 从 10 下降到 6。


```
top - 00:05:12 up 151 days, 22:17, 3 users, load average: 8.04, 3.82, 1.88
Tasks: 728 total, 3 running, 725 sleeping, 0 stopped, 0 zombie
Cpu(s): 36.6%us, 13.5%sy, 0.0%ni, 48.6%id, 0.0%wa, 0.0%hi, 1.2%si, 0.1%st
Mem: 12053504k total, 11764764k used, 288740k free, 366716k buffers
Swap: 2008116k total, 120k used, 2007996k free, 6227460k cached
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|-------|----|----|-------|------|-----|---|------|------|----------|---------|
| 2239 | admin | 16 | 0 | 5147m | 4.1g | 26m | R | 73.6 | 35.6 | 34:22.25 | java |
| 6924 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 7.6 | 35.6 | 0:03.02 | java |
| 9804 | admin | 16 | 0 | 5147m | 4.1g | 26m | S | 6.9 | 35.6 | 0:01.09 | java |
| 9815 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 6.6 | 35.6 | 0:03.01 | java |
| 6976 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 6.6 | 35.6 | 0:01.58 | java |
| 6920 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 6.3 | 35.6 | 0:01.16 | java |
| 9820 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 5.6 | 35.6 | 0:03.27 | java |
| 9826 | admin | 16 | 0 | 5147m | 4.1g | 26m | S | 5.6 | 35.6 | 0:02.40 | java |
| 6952 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 5.6 | 35.6 | 0:01.42 | java |
| 6956 | admin | 16 | 0 | 5147m | 4.1g | 26m | S | 5.6 | 35.6 | 0:01.11 | java |
| 6922 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 5.3 | 35.6 | 0:01.38 | java |
| 6958 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 5.3 | 35.6 | 0:03.52 | java |
| 6929 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 5.0 | 35.6 | 0:00.79 | java |
| 11555 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 4.6 | 35.6 | 3:02.09 | java |
| 6925 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 4.6 | 35.6 | 0:01.30 | java |
| 6940 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 4.6 | 35.6 | 0:00.88 | java |
| 6950 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 4.3 | 35.6 | 0:01.77 | java |
| 9806 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 4.0 | 35.6 | 0:03.02 | java |
| 9831 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 4.0 | 35.6 | 0:15.72 | java |
| 6979 | admin | 16 | 0 | 5147m | 4.1g | 26m | S | 4.0 | 35.6 | 0:01.83 | java |
| 11557 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 3.6 | 35.6 | 0:56.38 | java |
| 7244 | admin | 15 | 0 | 5147m | 4.1g | 26m | S | 3.0 | 35.6 | 5:18.53 | java |

图 4-23

```
"http-0.0.0.0-8080-2" daemon prio=10 tid=0x00002aaaca8bf000 nid=0xa67 in Object.wait() [0x0000000046922000]
"Thread"
    ;CallBackTasksScan" daemon prio=10 tid=0x00002aaac0119000 nid=0x8bf waiting on condition [0x0000000042863000]
```

图 4-24

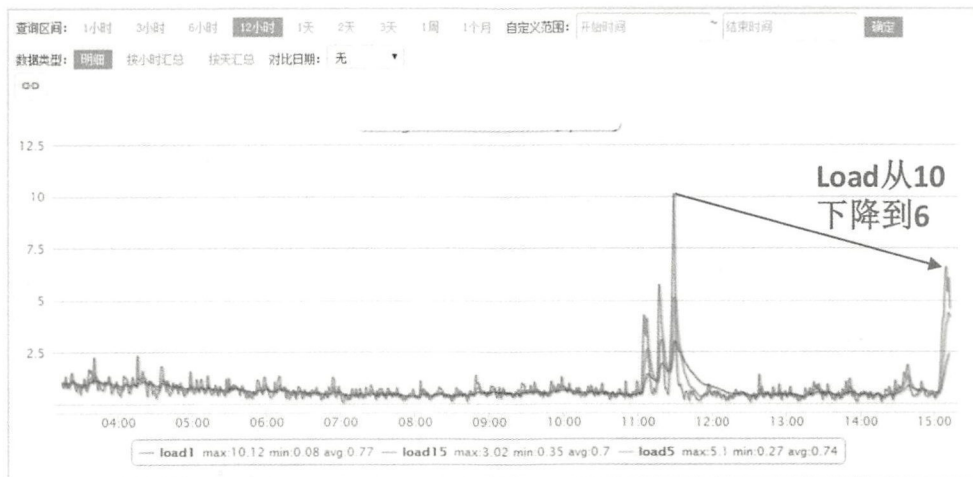


图 4-25

如图 4-26 所示，CPU 占用率下降了 25%。下降 25%是一个合理的表现，因为对于以 Tair 为主的逻辑而言，在 4 核的机器上，新代码解放了一个核，该核不用再死循环，所以 CPU 占用率降低了 25%。

但是，做一次实验是不够的，再来一次，看一看效果，如图 4-27 所示。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

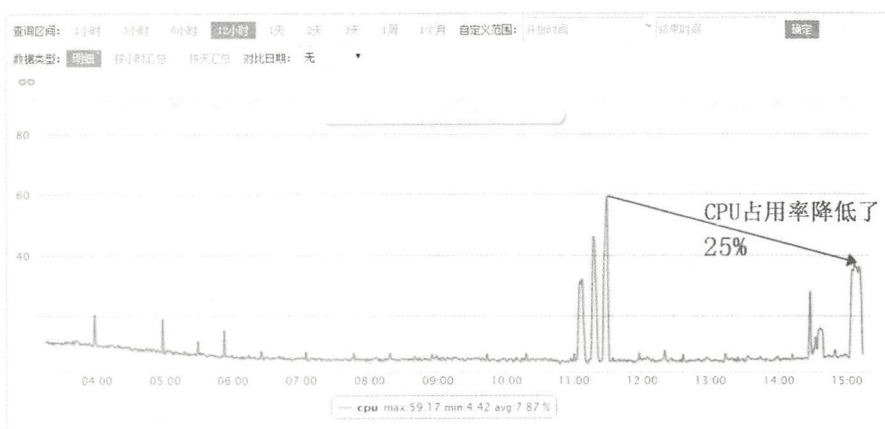


图 4-26

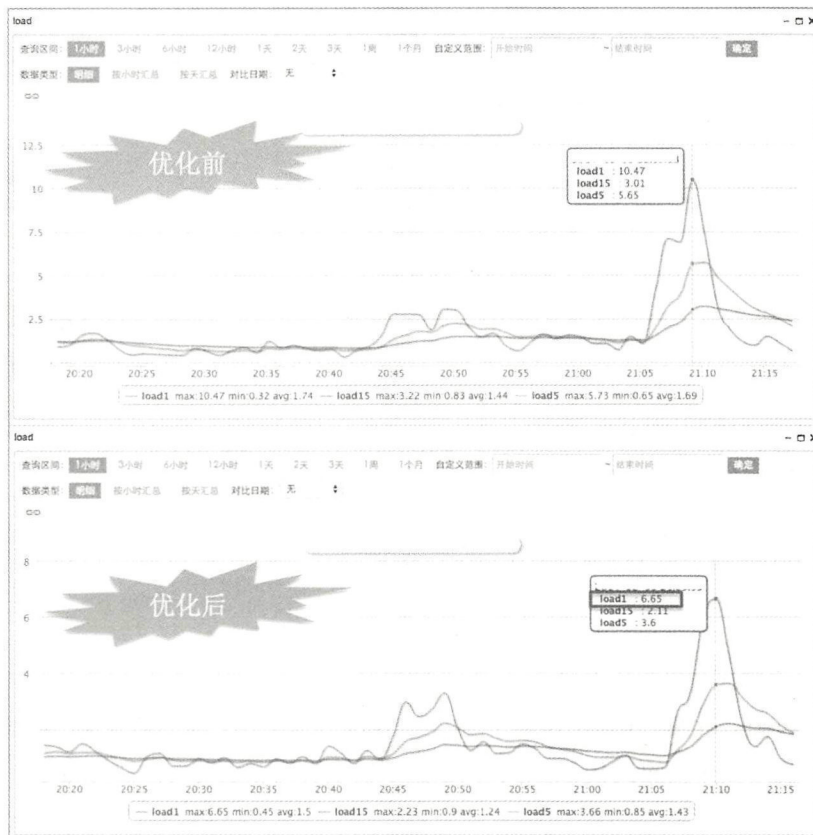


图 4-27



优化后的 Load 是 6.65，优化前的 Load 是 10.47，这次要看 Load1，不要看 Load5 和 Load15，因为这次压力测试只持续了 5 分钟。

然后来看 CPU，如图 4-28 所示。优化后的 CPU 占用率是 45.58%，优化前的 CPU 占用率是 63.85%。

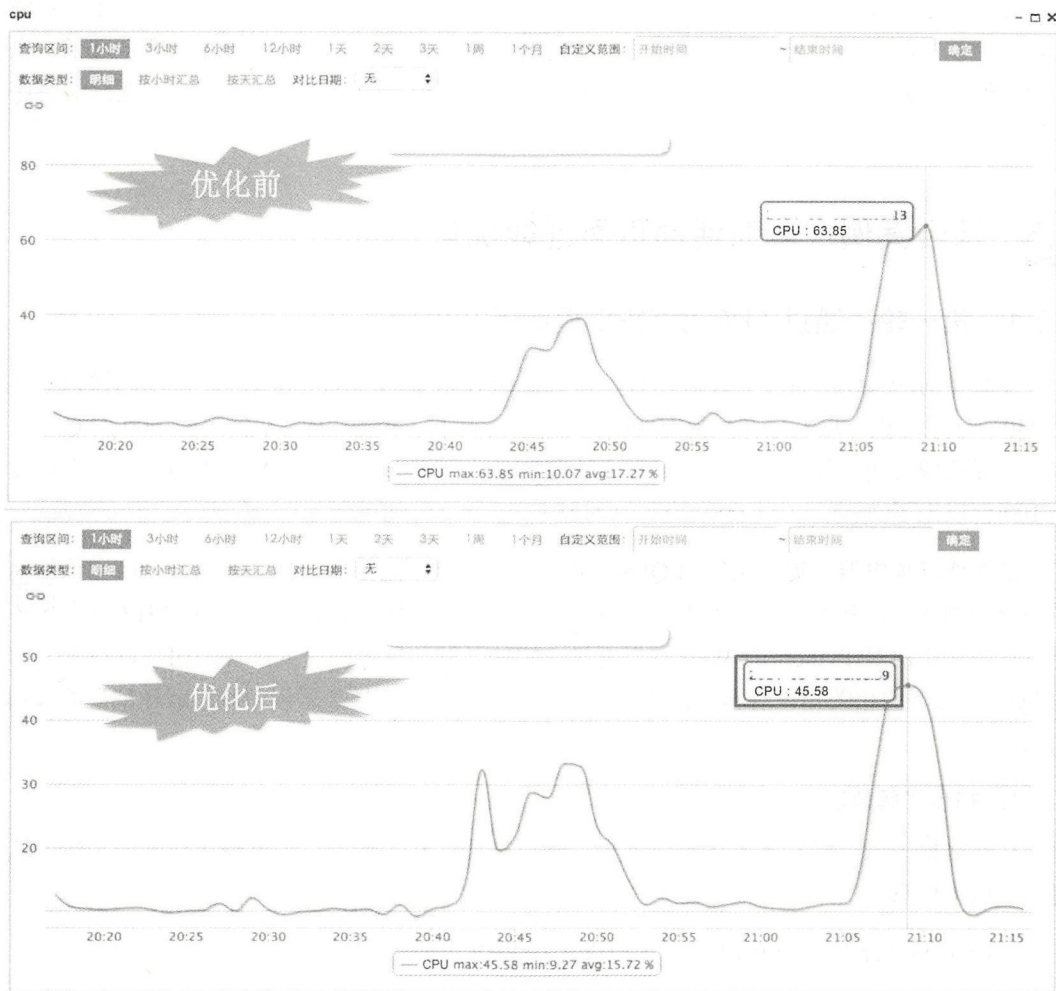


图 4-28

这次优化的效果非常明显，CPU 占用率基本能降低 25%，Load 也能下降不少。关键是这次优化惠及的系统非常多，某业务部门很多系统都依赖该中间件，而且很多系统都使用了 mget，





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

相信通过这样的优化，可以为该业务部门甚至全集团带来非常多的机器资源节约。最终笔者把这个问题的原因和解法告知了中间件团队，一段时间之后他们修复了该漏洞。

4.4.6 小结

这个问题的发现和解决过程比较简单，就是看代码、理逻辑。不像有些问题，尤其是网络问题，即使从浏览器发起请求开始梳理到浏览器再拿回数据，整条链路的环节都走过了，也不一定能解决问题。在查看代码的时候，需要梳理多个线程之间的交互关系及状态变化的时序，这样能够更快地发现问题。

4.5 综合案例：电商活动页面性能优化

4.5.1 第一轮：通过 APC 使 QPS 提高近 3 倍

在数年前的某次大促中，A 应用有多台机器的 Load 增至 100 多，还有一台机器的 Load 增至 240，导致宕机，这发生在集群有 30 台机器的情况下（30 台机器只能支撑 6000 的 QPS）。这次大促的成交额预计翻三倍，这样带来的问题是，如果不做优化，A 应用的机器量直接要增至 100 台以上了。

虽然通过加机器来提升系统的 QPS 是可行的，但这不是第一选择，加机器之前应该优化一下系统。刚开始，基本上无从入手，因为 A 应用是一个 PHP 容器，而笔者对 PHP 没有太多研究，只是几年前对 Python 的 Web 开发有过一定的了解，后来的研究过程证明，以前学习的那些 Python 部署的相关知识对本次优化起到了很大作用，尤其是部署模型和“解释型”语言相关知识。

1. 初步了解情况

从什么地方入手呢？笔者认为应该先了解 C 应用的相关情况。经过较为详细的了解，笔者得到了以下信息：

- （1）C Server 是一个 PHP 容器，提供运行 PHP 页面的功能。
- （2）这些 PHP 页面的体积非常大，活动页面的单个 PHP 文件基本上都在 100KB 以上。
- （3）PHP 文件由某个 CMS 系统生成，其中 CMS 系统的模块中定义了 PHP 常量，然后再通过 PHP 代码将这些常量赋值到 HTML 标签中相应的位置。
- （4）这里还有一个关键点。现有的 PHP 容器是 Apache+mod_php，奇怪的是，笔者使用的





是 Apache 的 worker 模型。为什么是 worker 模型？因为这里的 Apache 在编译的时候只选择了 worker 模型，Apache 平时是 Jboss 前面的代理服务器，在平时的场景中使用 worker 模型会更加节约机器资源。

我们看一下 C 应用下典型的 PHP 页面：

```
<?php
$naviCategory = array(
    1  -1 =>array('naviName'=> 'xx', 'href'=> '#guid-13942631976360',
'isShowNavi'=> ' '), 2  -1 =>array('naviName'=> 'ff', 'href'=>
'#guid-13941780271620', 'isShowNavi'=> 'true'), 3  -1 =>array('naviName'=>
'xd', 'href'=> '#guid-13956549694260', 'isShowNavi'=> 'true'), 4  -1
=>array('naviName'=> 'cc', 'href'=> '#guid-13941781344440', 'isShowNavi'=>
'true'), );

$options = array(
    1  -1 =>array('needFloat'=> 'true', 'floatRight'=> 'true',
'verFloatOffset'=> ' ', 'isGoTop'=> 'true'), );

$options = $options[0];
?>

<div class="ff" conf-float-disable="<?=$options['needFloat']?
'false':'true' ?>" conf-float-offsettop="<?=$options['verFloatOffset'] ?>"
<div class="hori-offset-box <?=$options['floatRight']?'right':'left' ?>-side">
    <div class="J-float-box float-box" style="margin-top:<?=$options['needFloat'] ?>">
        <div class="float-header"></div>
        <div class="float-bg">
            <ul>
                <?php foreach ($naviCategory as $k => $v) : ?>
                    <li class="<?php if ($v['isShowNavi']) : ?>
show<?php endif; ?>"><a href="<?=$v['href']; ?>"><?=$v['naviName']; ?>
</a></li>

                <?php endforeach; ?>
            </ul>
            <?php if ($options['isGoTop']) : ?><span class="go-
top"><a href="#"></a></span><?php endif; ?>
        </div>
    </div>
```





```
</div>  
</div>
```

比如，上面定义一个变量 `naviCategory`，赋值之后，在下面的 `div` 中使用。页面上绝大部分都是这样的逻辑（定义变量、赋值，再使用变量）。这是有明显的规律可循的，笔者把这样的页面称为“类”静态页面。也不完全是这样的页面，因为还有一些简单的逻辑在其中，最典型的就是根据 `user-agent` 来返回对应的 HTML 内容，比如为手机浏览器返回一段 HTML，为 PC 浏览器返回另外一段 HTML。面对这样的部署场景和逻辑，读者会想到什么解决方案呢？

2. 现状总结

C 应用非常简单，就是一些大的 PHP 文件（100~500KB），这些文件由 CMS 生成，不会对后台的任何 Server 执行远程调用，里面只有一些最简单的逻辑，比如根据请求客户端的类型来显示相应的数据，根据服务器端的时间来做相应的逻辑处理，而且这些逻辑在代码里出现得非常少，大部分页面就是 PHP 变量的定义和输出（输出到 HTML）。

那么从哪里下手呢？在这个环境中，既没有远程调用，也没有磁盘 I/O（通过 `iostat` 查看），更没有什么复杂算法或者业务逻辑。没有使用复杂的针对某些特定领域的数据结构，没有不常见的技术，为什么它的 Load 这么高？

3. 思路

早在 10 年前，笔者对 Python 有过一些了解，尤其是 Python 的 Web 开发框架 Django，那时经常拿 PHP、Python、Java 这三者在 Web 开发领域进行比较。PHP 开发的高效是早有耳闻的，而且 PHP 是“解释型”语言，那么 A 应用的问题会不会是部署模型导致的，或者是 PHP 本身的性能导致的？

所以笔者基本上确定了下面三个方向：

- （1）PHP 语言级别优化。
- （2）PHP 部署模型优化。
- （3）PHP 页面片段缓存。

为什么确定这三个方向？因为笔者有学习 Django 的经验，Django 上有篇文档描述了部署问题，以前笔者也做过一些测试。一开始的时候，笔者内心一直期望这次遇到的是部署调优的问题，因为这让笔者有一种熟悉感和安全感。随着研究的深入，笔者越来越有信心。

语言级别的优化是因为之前了解过 `pyc` 这种文件，本质上也是为了避免重复的“解释”所





带来的性能开销。所以笔者想知道 PHP 上是否也有这样的优化。

至于 PHP 的页面片段缓存，这在 Java 和 Python (Django) 中也是非常常见的，笔者当时的想法就是，这些重复 HTML 的渲染绝对是浪费，如果能节省，应该可以带来很大的性能提升（后面的测试证明，CPU Time 的主要消耗不在这里，所以这里的提升是非常有限的）。

4. 字节码缓存之 APC

1) APC 带来的效果

对于 PHP 自身来说，在不改造代码的前提下，能否用最小的代价来实现优化？于是研究开始。首先学习的是 PHP 语言的工作机制，上网查了一些资料，发现 PHP 会产生 OPCode 这样的中间代码，这和 Java 编译之后产生字节码的道理是一样的（理解字节码对程序员来说是非常重要的技能，研究 Kilim 的时候，就是通过 Kilim 产生的字节码了解了实现 coroutine 的方法）。还有一个重点是，这个 OPCode 在默认的 PHP 环境里，每次请求都需要重新产生 OPCode，这绝对会导致资源浪费。试想一下，Java 类每次执行之前都 Javac 一下，程序员们会有什么反应，他们一定会说：“真是坑啊。”

于是查了一下资料，发现 PHP 其实也是可以缓存 OPCode 的，而且业界有不少组件，其中比较出名的是 EACC 和 APC。于是笔者在自己的机器上安装了 EACC 和 APC，以及 PHP+Apache (prefork) 的环境。

值得注意的是，笔者请求的时候，使用的 IP 地址是 127.0.0.1，所以数据不会经过交换机，还是在本机流转，同时笔者使用 Apache Bench 测试，Apache 上也没有开启 Gzip，纯粹测试一下页面上 APC 带来的性能影响。

在这次大促的活动页面上，在笔者自用笔记本上，QPS 翻了 3 倍左右 (Apache+prefork+mod_php)，这也符合 APC 首页对它自己的介绍。

2) 扩展阅读：PHP 5.5 以上的 OPcache

需要大家注意的是，在 PHP 5.5 以上的版本中，已经自带了这样的功能，但是并非通过 APC 或者 EACC 实现，而是另外一个插件 OPcache。

OPcache 通过将 PHP 脚本预编译的字节码存储到共享内存中来提升 PHP 的性能，存储预编译字节码的好处就是，省去了每次加载和解析 PHP 脚本的开销。PHP 5.5 及后续版本中已经绑定了 OPcache 扩展。PHP 5.2、5.3 和 5.4 版本可以使用 PECL 扩展中的 OPcache 库。





5. PHP 片段缓存的实现

前面讲到根据页面的形态和特征，笔者决定在 PHP 页面上做片段缓存，但事实证明，在该场景下笔者走了一段弯路。

其实在使用 Nginx 之前，笔者在本机装了 Apache 和 APC，当时笔者注意到活动页面其实是一个“类”静态页面，其中大部分文本是可以缓存起来的，缓存到哪里呢？经过研究，笔者发现 APC 有 User Cache 这个东西，所以笔者选择将页面片段缓存到 APC 的 User Cache 中，在没有 Gzip 的情况下，运行效率提升非常大，因为几百 KB 的页面无须再次渲染，节约了大量的 CPU 资源，所以让人好兴奋，测试结果也让人非常兴奋，QPS 直接升至 1700。事后证明，加上 Gzip 之后，QPS 剧降。原因前面也提到了一点，那就是在笔者的场景下，PHP 生成 HTML 并不是消耗 CPU Time 的主要因素。

6. 部署模型优化

对于部署模型优化，之前对这种“解释型”语言的部署模型有过一点研究，比如 Python，可以使用 Apache+mod_python，也可以使用 Apache 或者 Nginx+fastcgi 的模型。那么 PHP 是不是也是这样的呢？查阅了一下资料，发现 PHP 基本也有这两种部署模型。读到这里，也许没有学过这些语言的读者会有点迷惑，不要紧，直接在百度上搜索 PHP、fastcgi、Nginx、Apache 之类的关键字，马上就可以找到相关的资料。如果懒得查资料，笔者也可以简单地描述一下这两种部署模型的差异，如表 4-7 所示。

表 4-7

| 比较维度 | Apache+mod_xxx | Nginx (Apache, lighttpd) + fastcgi |
|-------|--------------------------------|---|
| 进程 | PHP 在 Apache 进程中运行，Apache 内置模块 | PHP 有独立的进程，Nginx 和 PHP 进程以 fastcgi 协议进行通信，当然 Nginx 和 fastcgi 进程可以部署在不同的服务器上 |
| 额外的消耗 | PHP 在 Apache 进程中运行，无网络开销 | Nginx 和 PHP 进程还有一个网络通信，但是无须走交换机 |

对于这两者，笔者不发表评论，因为网上有很多测评，公说公有理，婆说婆有理，但是否是公平的测试，谁都不知道，有些人测试的结果是 Nginx+fastcgi 性能高，有些人测试的结果是两者性能相差不多，基本相同。

一开始，笔者优先选择的是在现有的环境下安装 APC。

1) 部署优化实施之在 Apache+mod_php 环境下安装 APC

于是准备在 Apache (worker)+mod_php 的基础上安装 APC，遗憾的是，搞了 3 天，环境





没有搞定, 怎么安装都无法让 PHP 正常加载 APC 模块, 同时考虑到 Apache (worker)+mod_php 并不是官方推荐的方式, 所以准备替换成 Nginx+fastcgi 的方式。

为什么多进程+进程内多线程的 worker 模型不是官方推荐的方式? 来看一看 PHP 的官方说明, 如图 4-29 所示。

Warning e do not recommend using a threaded MPM in production with Apache 2. Use the prefork MPM, which is the default MPM with Apache 2.0 and 2.2. For information on why, read the related FAQ entry on using Apache2 with a threaded MPM

图 4-29

2) 部署优化实施之替换成 Nginx+fastcgi

通过和其他部门同事的沟通, 笔者发现集团在一些地方使用了 Nginx+fastcgi 模型, 而且这个部署包里已经携带了 APC。于是把安装包拿过来, 安装并修改配置文件, 测试之后发现在同样配置的机器上, 在没有开启 Gzip 的情况下, 不经过路由器 QPS 上升到 1700。这里不是说 Apache+mod_php 比 Nginx+fastcgi 的性能差, 因为笔者本机的 Apache+mod_php 没有做任何优化。

当 QPS 上升到 1700 的时候 (这时候已经加了 PHP 片段缓存), 欣喜了 2 个小时。2 个小时之后, 仔细一看, 不对劲, 网卡流量已经超过了 1GB, CPU 还没有用满。这是为什么? Nginx 的 Gzip 是打开的, 突然想到 Apache Bench 测试里没有指定服务器是否可以接受 Gzip 的返回数据。于是重新测试, 在 Apache Bench 测试的命令上加 -H accept-encoding=gzip 即可。

3) 血泪史之 Gzip

加上 Gzip 之后 QPS 从 1700 跌到了 500 左右, 笔者感觉到一个明显的信号, APC 字节码缓存的优化到了尽头。而且去掉片段缓存之后, QPS 没有跌多少, 从 510 到 470 这种跌幅, 不影响大局。

我们知道, CPU Time、CPU 核数及 CPU 利用率决定了 QPS, 而在我们的场景下, APC 片段缓存带来的 CPU Time 的缩短在 Gzip 面前简直不值一提 (如果片段缓存中的业务逻辑很复杂, 那么片段缓存的优化效果会比较显著, 关键就看片段缓存生成所占用的 CPU Time 占整个 CPU Time 的比例, 比例越大, 片段缓存优化的效果越明显, 比例越小, 片段缓存优化的效果越不明显)。

7. 优化成果

即使走了弯路, 收获还是不小的 (虽然性能提升没有一开始那么大, 但也是因为这个曲折,





笔者找到了第二轮巅峰优化的“钥匙”）。开启了 Gzip, Apache (worker) + mod_php 的性能还不到 Nginx+fastcgi+APC 性能的一半, QPS 从 180 提高到了 510。也就是说, 原来 30 台机器的集群, 保守一点现在可以减少到 15 台。

值得注意的是, 上次大促的 A 应用的技术保障目标 QPS 是 6000, 用了 30 台机器, 而这次大促, 技术保障目标 QPS 是 12000, 但是 30 台机器轻松压到了 19000 的 QPS。虽然页面不完全相同, 但是基本是类似的。而且 Load 很低, 没有超过 8 台机器。

到这里为止, 优化的第一阶段已经取得了显著效果, QPS 提高了 2 倍多, 接近 3 倍。在这种情况下, 面对当前的大促, A 应用在现有机器的数量下可以说毫无压力。后来事实也证明, A 应用大促期间非常轻松。

8. 总结

整个过程是曲折的, 因为中间走了几段弯路, 比如很多时间花在了环境的搭建上。由于笔者的疏忽, 没有在 Apache Bench 上加压缩头, 导致白忙了一阵。而且由于前面的方向问题, 导致笔者花了很多时间在 PHP 的页面片段缓存的设计和压力测试上 (但其实也为第二轮优化埋下了伏笔)。即使走了这些弯路, 笔者觉得第一轮优化还是成功的, 因为有以下收获:

- PHP 也是有字节码的。
- PHP 的字节码默认情况下是不缓存的, 每次都需要“解释”。
- PHP 5.5 以上将自带字节码缓存。
- 加上字节码缓存之后 QPS 提高 2 倍多, 为大促节约了一半以上机器资源。
- Gzip 在我们的场景下已然成为 CPU “杀手”。

除上面拿到的结果外, 还有两个重要发现, 一个是 Gzip 在活动页面中的 CPU Time 所占的比重, 另一个是利用 APC 实现 PHP 的片段缓存。Gzip 问题是第二轮优化的目标, 而片段缓存是第二轮优化方案的基础。可以说, 如果在第一轮优化中没有亲自实现一遍 PHP 的片段缓存, 那么面对 Gzip 的问题, 笔者可能会束手无策。

4.5.2 第二轮: 解决消耗 CPU 资源大户 Gzip

通过 APC 和部署模型的变化, QPS 提升了两倍多。但是之前线下没有开启 Gzip 的时候, 加装 APC 和实现片段缓存之后, 性能可以提高 9 倍, 为什么开启 Gzip 之后整体效果下降了, 只能提高 2~3 倍了? 原因就在于 Gzip。进行压力测试时笔者发现, 在开启 Gzip 的情况下, CPU 资源的消耗情况如图 4-30 所示。





| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|-------|----|----|-------|-----|------|---|------|------|---------|---------|
| 11575 | admin | 22 | 0 | 45836 | 11m | 988 | R | 69.8 | 0.1 | 6:25.16 | nginx |
| 11574 | admin | 25 | 0 | 45580 | 10m | 988 | R | 66.8 | 0.1 | 6:25.78 | nginx |
| 11578 | admin | 25 | 0 | 45564 | 10m | 988 | R | 65.8 | 0.1 | 6:25.49 | nginx |
| 11576 | admin | 25 | 0 | 45564 | 10m | 1000 | R | 64.4 | 0.1 | 6:30.34 | nginx |
| 5705 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.7 | 0.2 | 0:07.98 | php-cgi |
| 5769 | admin | 15 | 0 | 281m | 13m | 9.9m | S | 1.7 | 0.2 | 0:06.96 | php-cgi |
| 5782 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.7 | 0.2 | 0:07.20 | php-cgi |
| 5849 | admin | 15 | 0 | 281m | 13m | 9.9m | S | 1.7 | 0.2 | 0:07.25 | php-cgi |
| 5681 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.85 | php-cgi |
| 5767 | admin | 15 | 0 | 281m | 13m | 9.9m | S | 1.3 | 0.2 | 0:07.22 | php-cgi |
| 5776 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:08.17 | php-cgi |
| 5786 | admin | 15 | 0 | 283m | 15m | 10m | S | 1.3 | 0.2 | 0:07.14 | php-cgi |
| 5787 | admin | 15 | 0 | 281m | 13m | 10m | S | 1.3 | 0.2 | 0:07.57 | php-cgi |
| 5797 | admin | 15 | 0 | 281m | 13m | 9.9m | S | 1.3 | 0.2 | 0:07.21 | php-cgi |
| 5703 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.62 | php-cgi |
| 5707 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.57 | php-cgi |
| 5720 | admin | 15 | 0 | 281m | 14m | 10m | S | 1.3 | 0.2 | 0:07.69 | php-cgi |
| 5739 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:08.10 | php-cgi |
| 5741 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.49 | php-cgi |
| 5745 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.66 | php-cgi |
| 5749 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:08.18 | php-cgi |
| 5753 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:08.03 | php-cgi |
| 5758 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.45 | php-cgi |
| 5762 | admin | 15 | 0 | 281m | 13m | 9.9m | R | 1.3 | 0.2 | 0:07.29 | php-cgi |
| 5772 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:07.19 | php-cgi |
| 5779 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.3 | 0.2 | 0:06.96 | php-cgi |
| 5788 | admin | 15 | 0 | 281m | 12m | 9.9m | R | 1.3 | 0.2 | 0:07.05 | php-cgi |
| 5797 | admin | 15 | 0 | 281m | 13m | 9.9m | S | 1.3 | 0.2 | 0:07.17 | php-cgi |
| 5820 | admin | 15 | 0 | 281m | 13m | 9.9m | S | 1.3 | 0.2 | 0:07.38 | php-cgi |
| 5684 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.0 | 0.2 | 0:07.62 | php-cgi |
| 5686 | admin | 15 | 0 | 281m | 12m | 9.9m | S | 1.0 | 0.2 | 0:07.75 | php-cgi |

图 4-30

有时候 Nginx 消耗的 CPU 资源在 75% 左右, 图 4-30 显示在 70% 左右, 而一旦把 Gzip 关闭, Nginx 的 CPU 资源消耗立刻下降。

于是笔者花了更多的时间, 做了更多的测试。如图 4-31 所示是压缩级别对 QPS 和带宽的影响 (92KB、138KB、182KB、248KB、295KB 是页面大小)。

| 压缩级别从6到3 | | | | 压缩级别从6到1 | | | |
|----------|------|------|------------|----------|------|------|------------|
| 6->3 | QPS | RT | Band Width | 6->1 | QPS | RT | Band Width |
| 92KB | ↑51% | ↓32% | ↑13% | 92KB | ↑60% | ↓37% | ↑19% |
| 138KB | ↑53% | ↓37% | ↑12% | 138KB | ↑63% | ↓40% | ↑17% |
| 182KB | ↑60% | ↓45% | ↑13% | 182KB | ↑70% | ↓45% | ↑20% |
| 248KB | ↑65% | ↓39% | ↑16% | 248KB | ↑68% | ↓40% | ↑25% |
| 295KB | ↑61% | ↓38% | ↑16% | 295KB | ↑70% | ↓42% | ↑25% |

图 4-31





从图 4-31 中可以看到，压缩级别一下降，QPS 就提高很多，RT 也降低很多，但是页面大小会增加，从而导致带宽增加。

结合图 4-30 和图 4-31，可以看出 Gzip 是 CPU 资源消耗大户，如果降低一下压缩级别呢？于是又经过两天的实验，得到如图 4-32 所示的结果。

| 页面平均大小 200KB左右 | ■ 现有环境（默认压缩级别6） | ■ 默认压缩级别6 | ■ 降低压缩级别到3 | ■ 降低压缩级别到1 | ■ APC片段缓存+默认压缩6 | ■ APC片段缓存+压缩级别降到3 | ■ APC片段缓存+压缩级别降到1 |
|-------------------|------------------|--------------|------------|------------|-----------------|-------------------|-------------------|
| QPS | 170 | 500 | ↑700 | ↑724 | ↑600 | ↑900 | ↑1000 |
| Bind Width | 现有值（基准值） | 现有值（基准值） | ↑13% | ↑20% | 与现有值持平 | ↑13% | ↑20% |
| RT | 高于基准值3倍（300ms级别） | 基准值（100ms级别） | ↓未测 | ↓未测 | ↓22% | ↓45% | ↓45% |
| 页面大小 | 现有值（基准值） | 现有值（基准值） | ↑13% | ↑20% | 与现有值持平 | ↑13% | ↑20% |

图 4-32

把 Gzip 关闭或者降低压缩级别会提高带宽消耗，经过详细的压力测试（对不同大小的页面，通过不断地调整压缩级别）和计算，降低压缩级别带来的 QPS 的提高所节约的机器费用和带宽提升所带来的费用提升不相上下（机器按照 3 年折旧来算），由于某些数字比较敏感，所以这里就不给出详细的计算公式了。

降低压缩级别的方法看上去很有效，但是却没办法操作。在这个场景下 Gzip 又是消耗 CPU 资源最大的用户，我们貌似进入了僵局，优化看上去已经结束了，因为 Gzip 是避不开的。如果不压缩，带宽的消耗会损失很多钱。

那么优化是不是要停止呢？笔者没有放弃，笔者的目标是成为“能工巧匠”。于是继续想，忽然间灵感来到，有了一个新的想法，即用 PHP 把 HTML 预先压缩好，放到内存里。由于之前笔者走了片段缓存的弯路，能否用片段缓存的思路来存储预先压缩的数据？于是一个方案在笔者心中成形。

笔者发现 Gzip 在活动页面对性能的影响非常大，因为活动页面没有复杂的计算逻辑，所以消耗 CPU 资源的大户只有 Gzip。而且笔者还发现，在我们的场景下 Gzip 已经成为待解决的最大问题，于是笔者花了一段时间，对活动页面的场景做了一系列降低压缩级别的测试，测试结果如图 4-33 所示。



| 压缩级别从6到3 | | | | 压缩级别从6到1 | | | |
|----------|------|------|------------|----------|------|------|------------|
| 6->3 | QPS | RT | Band Width | 6->1 | QPS | RT | Band Width |
| 92KB | ↑51% | ↓32% | ↑13% | 92KB | ↑60% | ↓37% | ↑19% |
| 138KB | ↑53% | ↓37% | ↑12% | 138KB | ↑63% | ↓40% | ↑17% |
| 182KB | ↑60% | ↓45% | ↑13% | 182KB | ↑70% | ↓45% | ↑20% |
| 248KB | ↑65% | ↓39% | ↑16% | 248KB | ↑68% | ↓40% | ↑25% |
| 295KB | ↑61% | ↓38% | ↑16% | 295KB | ↑70% | ↓42% | ↑25% |

图 4-33

同时由于降低压缩级别，导致压缩之后的页面大出了几 KB，极有可能导致在网络通信时 RT 中增加新的 RTT，也许这在国内的网络环境中影响不是特别大，但是对于世界范围内的网络环境，一个 RTT 有可能达到数百毫秒，对 RT 还是有一定影响的。这一点，后面还会阐述。

节约机器的钱和带宽增加带来的成本投入是差不多的。而且降低压缩级别导致的 RT 上升也对国际友人的用户体验不好，所以降低压缩级别在这样的场景里也是不可靠的。

1. 思路和问题

如果压缩是不可避免的，那么怎么做优化呢？能否将压缩提前做好呢？如果提前压缩好，放在内存中，用户请求通过的时候直接返回内存中的数据，岂不妙哉？直接把 CPU 密集型应用改造成了 I/O 密集型应用，这应该很有意思。

看起来很好的一个方案，但还是得好好研究一番：

- PHP 是进程模型，压缩过的数据应该放哪里？
- 如果是预先压缩，打点怎么搞？
- 这段 PHP 代码应该怎么写？
- 怎么告诉 Nginx 不需要再压缩了？
-

这些问题不解决，就无法继续前进，那么预先 Gzip 这个方案看上去就无法在该应用上实施了。

2. 解决方案

1) 压缩后数据放哪里



- 放进程中？

由于 PHP 是进程模型，为了提高并发处理能力，通常会开很多个进程，如果有一千个页面，每个页面消耗 20KB 的内存，开了 40 个进程，那么总内存消耗变成了 $20\text{KB} \times 1000 \times 40 = 800\text{MB}$ 。每个进程都存储了相同的页面数据。如果在 Java 里，只需要 20MB 的消耗。所以一个 20MB 空间的需求活生生地弄成了 800MB，这是广大程序员所不能接受的。

- 放分布式缓存中？

没办法，那怎么解决这个问题呢？有人说放到分布式缓存里，但是使用分布式缓存来存储也有几个问题，我们来看看有哪些问题，这些问题都是已经遇见过的：

- 分布式缓存网卡流量有可能跑满，某次大促某个应用即使只放了 2KB 的数据，由于 QPS 高，也将分布式缓存网卡流量跑满了，不得不先扩容，因为分布式缓存的 Server 一般会比 Web Server 少很多，而且基本都是公用的，跑满还会影响其他服务。
- 每个请求 20KB 的数据，都要从 Tair 返回，影响了 RT。
- 无法迁移到海外 CDN 集群中，我们总不可能在 CDN 集群里部署分布式缓存集群吧？

所以放到分布式缓存中是不可行的。

- 放磁盘中？

好像选择的余地也不是很多啊，想来想去，还有一个地方可以放，就是本机内存或者本机磁盘，但是放在本机磁盘中是否可行笔者不确定，现有的应用是不是 SSD，这些都是放磁盘的一些限制。或者用内存做磁盘镜像（tmpfs 之流），这样限制更少，但是带来一个问题，就是容量控制和运维的难度增加了，我们的方案要优先减少各方的工作量，尤其是工程师的工作量。所以将数据放入磁盘中不是最优方案。

- 放共享内存中？

放在共享内存中就简单得多，但是需要一个工具，这个工具能够让 PHP 把数据放到系统的共享内存中。最好是现有的 CDN 集群就支持，有这样的集群吗？

有的，就是 APC。前文中讲到，APC 可以将 PHP 代码的字节码缓存起来，但是笔者没有讲的是，APC 其实还有一个功能，叫作 User Cache。何为 User Cache？即可以把用户数据存储在这里的一种 Cache，而不只是 PHP 的字节码数据。

2) APC 的 User Cache

- 文档研究

首先研究 APC 文档，在连接中，有几个参数是跟 User Cache 相关的。



- `apc.shm_segments`: 编译器缓存要分配的共享内存块的数目。如果 APC 用光了共享内存, 但是已经将 `apc.shm_size` 设为了系统所能允许的最大值, 可以尝试增大此值。
 - `apc.shm_size`: 以 MB 为单位的每个共享内存块的大小。有些系统 (包括大多数 BSD 变种) 默认的共享内存块大小非常小。
 - `apc.user_ttl`: 缓存条目在缓冲区中允许逗留的秒数。0 表示永不超时。值设为 0 意味着缓冲区有可能被旧的缓存条目填满, 从而导致无法缓存新条目。如果大于 0, APC 将尝试删除过期条目。只是针对每个用户而言, 建议将值设为 7200~86400。
 - `apc.gc_ttl`: 缓存条目在垃圾回收表中能够存在的秒数。此值提供了一个安全措施, 即服务器进程在执行缓存的源文件时, 如果该文件被修改则旧版本将不会被回收, 直到此值为止。设为 0 将禁用此特性。
- APC 代码研究

值得注意的是, 由于线上使用的是 APC 3.0.9, 所以笔者看的是 3.0.9 版本的源代码。

APC 中 User Cache 的存储结构是一个典型的散列链表, 和 HashMap 的实现是类似的道理, 但是没有 HashMap 这么精致, 来看一段代码。

```
apc_cache_entry_t* apc_cache_user_find(apc_cache_t* cache, char *strkey,
int keylen, time_t t)
{
    slot_t** slot;

    LOCK(cache);
    /* Cache 里有一个 slot 的数组, 叫作 slots, 取模之后找到对应的 slot */
    slot = &cache->slots[string_nhash_8(strkey, keylen) % cache->num_slots];
    /* 找到 slot 之后, 拿到一个链表, 开始遍历这个链表, 这个结构和 HashMap 是一样的, 但是在
    取模的问题上, HashMap 有更巧妙的算法 */
    while (*slot) {
        if (!memcmp((*slot)->key.data.user.identifier, strkey, keylen)) {
            if ((*slot)->value->data.user.ttl && ((*slot)->creation_time +
            (*slot)->value->data.user.ttl) < t) {
                remove_slot(cache, slot);
                break;
            }
            (*slot)->num_hits++;
            (*slot)->value->ref_count++;
            (*slot)->access_time = t;
        }
        /* 这种代码看起来是不是很熟悉 */
        cache->header->num_hits++;
    }
    UNLOCK(cache);
}
```




```
        return (*slot)->value;
    }
    slot = &(*slot)->next;
}

cache->header->num_misses++;
UNLOCK(cache);
return NULL;
}
```

从上面这段代码中，我们基本得知了 APC 中 User Cache 的结构，下面来看看 APC 如何插入新值。

```
int apc_cache_user_insert(apc_cache_t* cache, apc_cache_key_t key,
apc_cache_entry_t* value, time_t t TSRMLS_DC) {
    slot_t** slot;
    size_t* mem_size_ptr = NULL;
    if (!value) {
        return 0;
    }
    LOCK(cache); process_pending_removals(cache);
    slot = &cache->slots[string_nhash_8(key.data.user.identifier,
key.data.user.identifier_len) % cache->num_slots];
    if (APCG(mem_size_ptr) != NULL) {
        mem_size_ptr = APCG(mem_size_ptr);
        APCG(mem_size_ptr) = NULL;
    }
    while (*slot) {
        if (!memcmp((*slot)->key.data.user.identifier,
key.data.user.identifier, key.data.user.identifier_len)) {
            remove_slot(cache, slot);
            break;
        } else
            if((cache->ttl && (*slot)->access_time < (t - cache->ttl)) ||
            ((*slot)->value->data.user.ttl && ((*slot)->creation_time +
(*slot)->value->data.user.ttl) < t)) {
                remove_slot(cache, slot);
                continue;
            }
        slot = &(*slot)->next;
    }
    if (mem_size_ptr != NULL) {
        APCG(mem_size_ptr) = mem_size_ptr;
    }
}
```



```
    } /* 如果不能创建 slot, 那么返回 0 */ if ((*slot = make_slot(key, value, *slot,
t)) == NULL) { UNLOCK(cache); return 0; } if (APCG(mem_size_ptr) != NULL)
{ value->mem_size = *APCG(mem_size_ptr); }
    UNLOCK(cache); return 1; }
```

代码中写道, 如果不能创建 slot, 那么就返回一个 0 告知用户这次缓存没有成功。同时我们可以看到, 用户在插入新值的时候, 需要遍历 slot 的链表, 根据 Cache 的 TTL 和 Cache 里的这个 slot 链表中所有元素的 TTL 找出可以被回收的空间。

这样的操作在 find 方法中也存在, 可以将其看作 APC 在执行 find 和 insert 操作时, 在对应的 slot 链表上根据 TTL 来做缓存的清除动作。这是 user_ttl 所起的作用。

当然 APC 在删除 slot 链表时还有一些逻辑, 根据源代码中的 remove_slot 方法, 在 remove 时, 如果 ref_count 小于等于 0, 那么直接释放这个 slot, 如果 ref_count 大于 0, 但是满足了 TTL 相关的时间条件, 那么就会将这个 slot 放到一个 deleted_list 中, 供 APC 中的 GC 来回收这个 slot 对象。这就是 gc_ttl 这个参数的作用: 控制 slot 在 deleted_list 中存活的时间。

- APC 中 User Cache 的总结

APC 的缓存清空是跟 TTL 相关的, 而不是 LRU, 所以先进缓存的数据, 即使没有人使用, 不到时间不会被清除, 这会导致先进缓存的数据在缓存过期之前一直在缓存中, 所以 user_ttl 时间不要设置为 0, 且 gc_ttl 也要大于 0, 这样长时间不被访问的页面会被清出缓存, 这是不错的选择。之所以要把 APC 代码拿出来, 看看它的缓存清空策略, 其中一个非常重要的原因是笔者怕它是 LRU (Least Recently Used), 如果是 LRU 的清空策略, 我们就必须更加小心, 在共享内存不多且访问比较平均的情况下, 有 LRU 命中率低的可能性。因为刚刚放进去的页面, 有可能因为 LRU 被清掉。

如果内存不足, APC 会返回失败, 告知 PHP 进程。也就是在最差情况下, 当共享内存不够时, 页面就得不到缓存, 那么就需要每次都做 Gzip, 这个最差的结果和目前的情况是一样的, 也就是说最差也不过就是回到现状, 只不过打点的工作需要 PHP 代码来实现了, 这还是可以接受的。所以笔者决定用 APC 来存储压缩后的 HTML 页面。

3) 重复压缩

由于返回的 HTML 已经被 PHP 压缩过了, 那么 Nginx 或者 Apache 再压缩一遍其实是浪费, 而且不光是浪费, 在 Firefox 下, 重复压缩的数据还不能正常显示。我们不能简单粗暴地关闭 Nginx 压缩, 因为不使用这套方案的 PHP 页面或者 Nginx 后面的其他进程, 比如 Node.js 之类的, 还是需要用到 Nginx 压缩的, 最好的方案就是在返回端有一个标识, 有了这个标识之后,



Nginx 就不再压缩返回数据，而且不影响浏览器显示。

- `gzip_types`

笔者可以在 `gzip_types` 上做点设置吗？可以，比如，只要返回 `content-type=text/plain` 就不执行压缩。普通的 PHP 页面没有使用指定的 `content-type`，默认使用 `text/html`，并且默认会压缩，这也不失为一种方案。

- `gzip_min_length`

比如压缩过的页面都是小于 50KB 的，那么可以设置为大于 50KB 才压缩，小于 50KB 不压缩。

4) 打点

由于打点依赖 Nginx，而通过 Nginx 时，数据已经被执行过 Gzip 了，所以 Nginx 不会再做反 Gzip，打点再 Gzip 这种事情，那么打点这件事情就需要交给程序来做了。这话说起来很轻松，但是实际上，这里是最麻烦的，要找到解决方案，不得不先把情况了解清楚。

- 预压缩的打点方案一，Cookie 传递 `time` 属性。
- 预压缩的打点方案二，分段压缩。

使用 Cookie 传递打点需要的 `time` 属性是满足现状的，但是后面如果打点迁移到新的打点工具（属性很多，不只是 `time`），那么需要存放在 Cookie 里的值会很多，维护管理将不太方便，所以使用分段压缩是一个比较好的选择。

也许你会想，那很简单，直接把打点前的 HTML 压缩成一个 Gzip 流，然后将打点数据压缩一下，最后压缩一下打点数据之后的 HTML，这样就可以压缩成 3 个完整的 Gzip 文件，返回给浏览器，这样做是最简单、最省力、最省事的。没错，笔者开始也是这么想的，但是在不断查资料的过程中，发现 HTTP 规范中明确指出，返回给浏览器的应该是整段的 Gzip 文件。

即使使用 `varnish` 中的 ESI 实现，对于 PHP 来说也是行不通的，除非自己写压缩扩展。

所以，在 PHP 上只能使用 Cookie 传递 `time` 属性的方案进行优化。

5) 这段代码应该怎么实现

- 流程图

要写代码，先定流程，所谓谋定而后动，所以笔者就画了一张流程图，如图 4-34 所示。

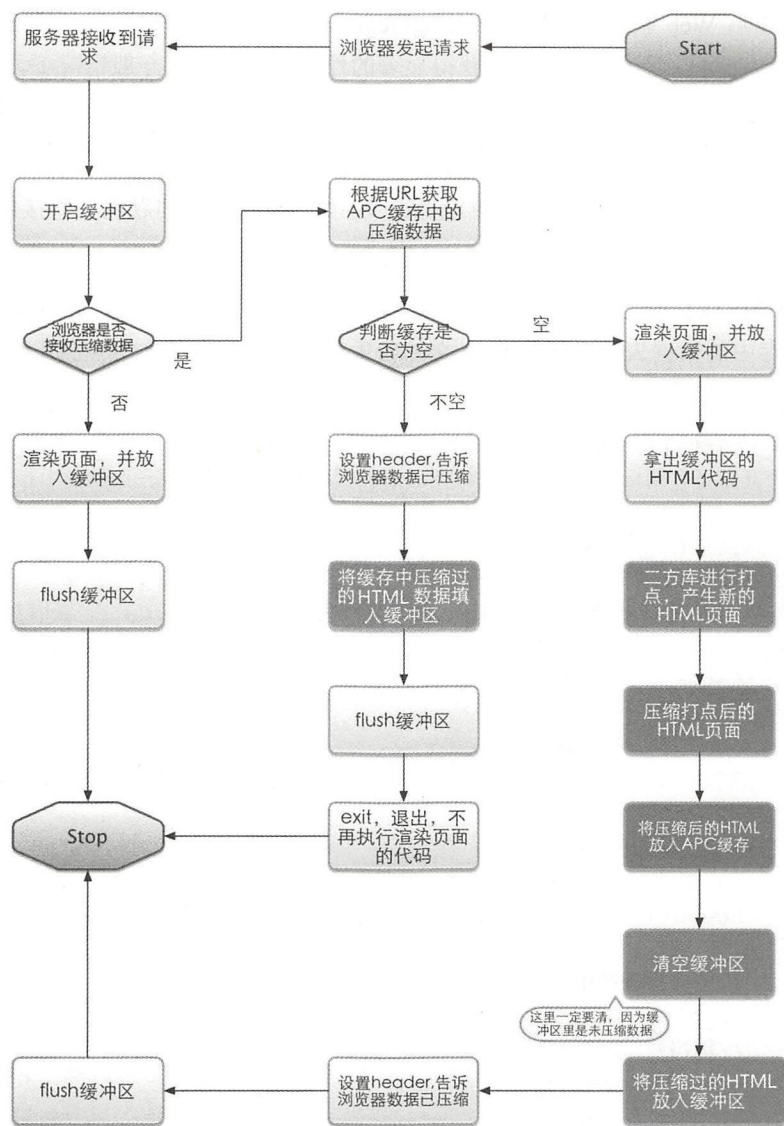


图 4-34

实际上 APC 3.0.9 有一个 `stats` 配置，改变这个值要非常小心。默认值 `On` 表示 APC 在每次请求脚本时都检查脚本是否被更新，如果被更新则自动重新编译和缓存编译后的内容，但这样做对性能不利。如果设为 `Off` 则表示不进行检查，从而使性能大幅提高。但是为了使更新的内



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

容生效，必须重启 Web 服务器（如果采用 cgi/fcgi 类似的进程，需重启）。生产服务器上的脚本文件很少更改，可以通过禁用本选项获得显著的性能提升。不过一般情况下，检查文件是否是最新的并不是性能瓶颈所在，Gzip 才是。所以建议大家设置成 `stats=on`，这样 PHP 文件更新时可以立刻自动重新编译，并缓存编译之后的内容。这一点非常重要。

如果页面改变了，但是缓存中的数据没有改变，如何解决这个问题呢？直接修改缓存的 key 即可，这样能保证最新的数据生效。

如果不想修改缓存的 key 呢？那么就需要将缓存时间设置得短一点了，比如 5 分钟，这样 5 分钟之后缓存中的数据失效，新的 PHP 文件就生效了。为了性能，这点付出也是必要的。

• Demo 代码实现

```
<?php ob_start();//缓冲区开始
//定义一个开始缓存的标识，这个函数将在后面的代码中被调用到 function 中
cache_start_apc() {
//如果客户端接收 Gzip 数据
if (canBeGzip()) {
    //根据 URL 生成一个 User Cache 的 key，并从 APC User Cache 获取对应的值
    $key = sha1($_SERVER['REQUEST_URI']);
    $content = apc_fetch($key);

    // $content = none;
    //如果缓存中该 key 对应的 value 不为空，那么直接返回缓存中的数据，否则退出函数，开
    始渲染页面
    if (!empty($content)) {
        header('Content-Encoding: gzip');
        header("Vary: Accept-Encoding");

        //清除缓冲区中的任何内容
        ob_clean();
        //输出数据到缓冲区，并执行 flush
        echo $content;
        ob_end_flush();
        exit;
    }
}
//如果客户端不接收 Gzip 数据，那么直接往下执行，渲染页面
}
//如果浏览器接收压缩数据，那么使用 zlib 库进行压缩，压缩级别为 6
function ob_beacon_and_gzip($content) {
```





```
//TODO 先打点，再压缩，打点模板需要改一下，把 time 改成变量
return gzencode($content, 6);
}
function cache_end_apc() {
//如果客户端接收压缩数据，则返回压缩数据，如果不接收压缩数据就不需要压缩并缓存了
if(canBeGzip()) {
    //从缓冲区中拿到渲染好的 HTML，并进行压缩
    $content = ob_beacon_and_gzip(ob_get_contents());
    //压缩后放到 User Cache 中，就算失败、内存不够也没有关系，直接返回压缩后的数据
    $key = sha1($_SERVER['REQUEST_URI']);
    apc_add($key, $content);

    //清空缓冲区
    ob_clean();
    //设置压缩头
    header('Content-Encoding: gzip');
    header("Vary: Accept-Encoding");

    //压缩内容输出
    echo $content;

    //flush 缓冲区
    ob_end_flush();
} else {
    //如果客户端不接收压缩数据，则把缓冲区中的原始 HTML 直接返回
    ob_end_flush();
}
}
function canBeGzip() {
return !headers_sent() && extension_loaded("zlib")
&& strpos($_SERVER["HTTP_ACCEPT_ENCODING"], "gzip");
} ?>
```

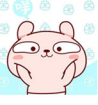
- 一定要 5 分钟之后生效吗

当然不一定，尤其现在的大促活动页面都是定制的，如果有紧急信息发布，只需要在修改时将缓存的 key 改一下即可，比如，原来 APC User Cache 中存储的 Gzip 对应的 key 是 123，那么在紧急发布时，新页面的 key 是 456 即可，原来的压缩数据在 5 分钟之后会被放入 GC 队列，然后等待被 GC 回收。

- 很担心 PHP 的压缩效率

不用担心，PHP 的压缩和 Nginx 的压缩都调用相同的库，都使用 zlib 库，而且如果将页面





全部缓存，同一个页面几分钟才需要做一次压缩，所以 PHP 的执行效率在这个场景下是不用担心的。

3. 测试

笔者尝试在不同的压缩级别和不同的页面大小的情况下做测试，并观察 CPU 和 Load 的情况，测试脚本如下：

```
**ab -n 10000 -c 10 -H 'Accept-Encoding:gzip' http://localhost:8888/xxx.php**
```

这些页面都来自数年前某次大促的真实页面，压缩级别=6 时测试结果如表 4-8 所示。

表 4-8

| 原始页面大小 | 压缩后的大小 | 优化后的QPS | RT |
|--------|--------|---------|-------|
| 92KB | 17KB | 2024 | 4.9ms |
| 138KB | 8.7KB | 1859 | 3.3ms |
| 182KB | 11.4KB | 2083 | 4.8ms |
| 248KB | 32KB | 1977 | 5.0ms |
| 295KB | 34.4KB | 1722 | 5.8ms |

整个测试没有经过网卡，而且是在一台使用多年的 MacBook Pro 上测试的：双核 8GB，三星的 SSD。

在把压缩级别调高之后，295KB 的页面压到了 33.9KB，和 34.4KB 没有太大区别，所以对 Gzip-level=9 没有进行更加深入的测试。

根据之前与 4 核虚拟机的对比来看，笔者预估：同样的程序如果放到 4 核虚拟机上，除去网卡带宽等限制不计，QPS 达到 3000 以上是没有压力的。

从以上测试结果来看，有几个结论比较抢眼。

(1) QPS 升得很高，高出现有的 10 倍左右。没有缓存压缩数据时 QPS 大多在 100~200 之间。

(2) RT 下降得很厉害，相对值很高，但是绝对值不高，下降范围在 50ms 以内。

(3) 压缩级别调高时：

- 带宽消耗降低，但不是特别明显。
- 由于包数量变少，所以假设 MTU=1500，MSS=1460，window size = 16328（笔者连英





文的 Amazon 时, window size 是 16328, 所以拿这个值举例)。如果页面 Gzip level = 9, 那么压缩完, 数据量小于 16328, 俄罗斯朋友会开心, 因为我们会一下子发 16328/1460 = 12 个包过去, 在理想情况下一个 RTT 内, 俄罗斯朋友就拿到了商品数据。如果 Gzip level = 6, 那么压缩完, 数据量有可能大于 16328, 那么我们就只发 12 个包过去, 在理想情况下等 12 个包的 ACK 最大 seq 的那个包返回, 再发剩余的包。这个时候就不是一个 RTT 的问题了。在国际网络环境下, RTT = 200ms 也是有的。当然这些都是估算, 针对我们的场景, 具体能出现什么样的优化效果, 也是需要长期测试的。而且一旦增加了海外集群, RTT 有可能达到 10ms, 这种提升压缩级别的效果就不明显了。

4. 能否使用 Web Cache 来缓存压缩之后的数据

如果以下这两个条件有一个不满足, 那么就无法使用 Web Cache 来缓存压缩之后的数据。

第一, 需要在 CDN 集群上部署 Web Cache, 现在是没有的, 不过部署起来也不是难事。

第二, 对于 PHP 代码中出现根据 user-agent 等 header 属性决定显示什么样的 HTML 来说, 这个需求直接用代码压缩的方案来实现就很方便了, 根据 user-agent 中的部分核心属性 (什么是部分核心属性, 因为 user-agent 太多了, 如果每个 user-agent 都作为一个 key, 则同一个页面会产生大量的副本, 对需求来说只是为了分辨出是 Mobile 还是 PC, 所以只要为数不多的几种 key 而已, 而且每种 key 对应的 HTML 也是不一样的, 不存在同一个页面有不同副本的问题), 渲染出不同的 HTML, 然后压缩并通过不同的 key 缓存在共享内存中。就好像这个需求和方案是天生一对一样, 如果用 SWIFT 来缓存不同 user-agent 的页面, 同一个 PHP 页面, 将会产生很多份缓存, 这样热点就不明显了, 命中率也会受到影响。因为在 Web Cache 中间件上是完整的 user-agent 来做 key 的一部分的, 所以 user-agent 越多, 副本就越多。

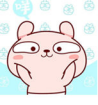
后来 CDN 相关同事也提到了这件事情, 确实 vary:user-agent 会产生大量的副本, 虽然 CDN 相关同事说的是会影响命中率, 但是笔者认为, 不光影响命中率, 还会影响热点集中度, 对于有多层 Cache 的缓存中间件来说, 热点集中与否直接影响页面在哪一层 Cache 上, 从而影响页面的响应速度。

4.5.3 小结

在这个优化中, 我们研究了 APC 相关的实现, 整理了整个流程, 并且用代码进行了实现, 唯一不完美的地方是打点, 目前只能把 time 属性放在 Cookie 中。

下面对比一下优化前后的两种方案, 如表 4-9 所示。





大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

表 4-9

| 各维度 | 优化前 | 优化后 |
|---------|-----------|--|
| TPS | 100~200左右 | 2000以上，在自己的老掉牙的笔记本上 |
| RT | 60ms以上 | 10ms以下 |
| PHP即时生效 | 即时生效 | 5分钟之后生效，如果在TMS中可以随机生成缓存的key，那么也可以做到即时生效，老的缓存数据让其自动过期 |
| 代码侵入 | 无代码侵入 | 少量代码侵入，把埋点代码向应用迁移 |
| 额外内存消耗 | 无额外内存消耗 | 有额外内存消耗（压缩后20KB的页面有1000个，需要20MB的共享内存） |
| 压缩级别 | 不能改变压缩级别 | 可以增加压缩级别，降低带宽消耗和RT，提升用户体验，但在上了海外CDN集群之后，Gzip level调整的必要性不高 |

如果只有 20 台机器，那么优化后只需要 3~5 台，我们可以不在乎，增加机器不是问题。如果有 100 台机器，保守估计可以优化到 30 台以内，乐观估计 15 台也不是没有可能。这时，少量的改造带来的就是大量机器成本的节约，即投入不大，但是产出是很大的。

虽然写得差不多了，但是笔者还要啰唆几句，这里预先 Gzip 只是一个优化思路，对和 PHP 无关的读者有什么助益呢？笔者简单列一下：

（1）知道在一些场景下，Gzip 可能是消耗 CPU 资源的大户，大家可以观察一下自己的应用。

（2）预先把浏览器需要的数据压缩之后放入缓存会带来 QPS 的极大提高（多高？笔者这个场景是 10 倍，取决于 Gzip 在整个 CPU Time 中的比重，读者的场景未必有这么多，也有可能更多）。

（3）虽然笔者是预压缩的 HTML，但是不代表不能压缩 Ajax 返回的 Json，对应返回的 Json 数据超过 100KB 的，每次都压缩一下 Json 和把 Json 压缩完放在内存的效率就不再多阐述了。

- 100KB 压完之后，只有 17KB 左右，内存占用少。
- 长时间内，只压缩一次，CPU Time 占用很少，提高 QPS。
- 没有打点的问题，操作起来非常简单。





5

第 5 章

TCP 优化

TCP 优化在传统的性能优化领域中很容易在应用层面被忽略，通常在 CDN 加速服务中会考虑这个优化。大型网站不仅有图片还有动态请求，经过 TCP 层面优化，可以让网络耗时变短。而在 CDN 层面，由于大型网站还有大量的图片和 JS 等资源，这些静态资源占整个页面请求的 90% 以上，所以只要性能提升 10%，整体的性能体验改观就相当明显。

特别是提供全球化服务的大型网站，近几年发展非常迅速，买家分布越来越广泛。跨境最大的挑战是地理距离长，网络延迟天然巨大，例如北美洲到欧洲的网络距离 RTT 在 250ms 以上。亚洲到北美洲的网络距离 RTT 在 200ms 以上。根据数据初步估计，南美洲的某个国家到达北美洲的美国的请求丢包率高达 6%~10%。一旦丢包，大部分都会发生超时重传，基于 3 次重复 ACK 的快速丢包发现算法，在很多情况下没有起到作用。减少丢包时的网络延迟，对跨境业务来说，比淘宝在国内得到的效益将更加可观。TCP 协议栈优化涉及的地方非常多，从增大初始拥塞窗口到减少默认的 RTO、PRR、Early Transmit，本章一一列举出来，目的只是做个总结，减少网络延迟。我们要关注这些技术，知道这些技术能够帮我们做什么事情非常重要。做必要的内核升级就可以减少网络延迟、提升性能。





5.1 TCP 传输原理

TCP 通过“发送—应答（ACK 确认）—重传机制”来确保传输的可靠性，它是端到端进行传输的。对于大型网站，响应报文从服务器端给客户端浏览器进行报文的传输，所以在服务器端，可以通过优化来降低响应给客户端的网络耗时。传统的 TCP 底层实现，在很多时候会导致 TCP 传输效率变低，特别是网络带宽的扩充和整体网络硬件技术的提升，使得网络传输速度有很大的变化。本节将简要介绍 TCP 传输的基本原理，以便对 TCP 优化知识进行了解。

5.1.1 TCP 传输的简要说明

TCP 传输是分段的，一个 HTTP 响应报文会被操作系统切成多个 MSS 大小（一般为 1460 B）的段，发送端每次只会发送若干段。能够发送多少个数据包，由拥塞窗口和接收端窗口共同决定，直到接收端接收到完整的报文为止。在此过程中，报文分段按照顺序进行发送，每个报文段在发送时，会做顺序编号，以便能够完整正确地组装，所以当 HTTP 的请求响应模型将请求发送给服务器时，服务器响应都需要多个 RTT 的传输，物理距离越远，总体网络耗时越长。

例如，如果大型网站的机房位于美国，用户在中国，而中美物理距离可能在 10000km 以上，那么用户端的一个数据包到达美国机房就需要 30ms，加上中间路由设备的转发延迟，以及 BGP 各个运营商之间的绕路（会在“CDN 优化”一章中介绍），网络耗时 150~200ms，有时甚至超过 200ms。每个报文发送多少段，就是由 TCP 的底层拥塞控制算法来进行控制的。报文越大，受拥塞控制算法的影响也越大，这也是本章可以重点优化的地方。对于大型网站来说，一般有很多页面，这些页面经过 HTTP 的压缩之后，仍然会高达数十 KB，甚至数百 KB。而当远距离传输时，一个数据包的来回网络耗时少则数十毫秒，多则数百毫秒，优化空间更大。

5.1.2 滑动窗口——接收端流量控制

滑动窗口本质上是描述接收方的 TCP 数据报缓冲区大小的数据的，发送方根据这个数据来计算自己最多能发送多长的数据。如果发送方收到接收方的窗口大小为 0 的 TCP 数据报，那么发送方将停止发送数据，等到接收方发送窗口大小不为 0 的数据报的到来。

关于滑动窗口，介绍 3 个术语。

- 窗口合拢：当窗口的左边缘向右边缘靠近的时候，这种现象发生在数据被发送和确认的时候。





- 窗口张开: 当窗口的右边缘向右边缘移动的时候, 这种现象发生在接收端处理数据以后。
- 窗口收缩: 当窗口的右边缘向左边缘移动的时候, 这种现象不常发生。

TCP 就是用这个窗口, 慢慢地从数据的左边缘移动到右边缘, 把处于窗口范围内的数据发送出去(但不用发送所有数据, 只是处于窗口内的数据), 这就是窗口的意义。

5.1.3 拥塞窗口——发送端流量控制

接收端流量控制在局域网内做流量控制是可行的, 但是在公网上就会出现问題。网络数据包在传输过程中, 要经过很多路由器的转发, 而这些路由器的带宽和缓冲区大小是不确定的。路由器的缓冲区太小, 会造成数据包大量堆积甚至拥塞, 而接收端的缓冲区一般很大, 此时会造成大量的网络拥塞, 从而加剧整个网络的拥塞, 甚至造成整个互联网不可用。为了解决这个问题, TCP 发送方需要确认连接双方线路的数据最大 QPS 是多少, 这就是所谓的拥塞窗口。

拥塞窗口的原理: TCP 发送方首先发送一个(或者数个)数据报文段, 然后等待对方的回应, ACK 回应后就把这个窗口的大小加倍, 然后连续发送两个数据报, 对方回应以后, 再把这个窗口加倍, 这个机制就是发送端拥塞控制的慢启动算法。

对慢启动算法的简单理解: 先发送少量的数据报文段, 得到确认后, 再将发送报文段的个数增加, 直到出现超时错误或者丢包; 发送端因此了解到网络的承载能力, 也就确定了拥塞窗口的大小, 发送方就根据这个拥塞窗口的大小发送数据。在下载文件时, 一般开始比较慢, 后面会慢慢变快, 直到达到一个稳定的速度。

慢启动的过程如图 5-1 所示。

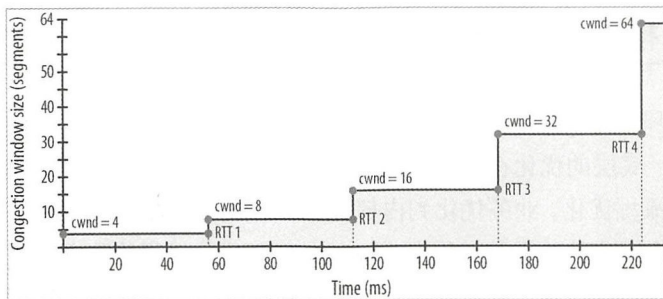


图 5-1

拥塞避免: 当发生超时没有得到对方的确认时, 发送的报文段个数会线性增长, 这就是拥塞避免阶段, 如图 5-2 所示。



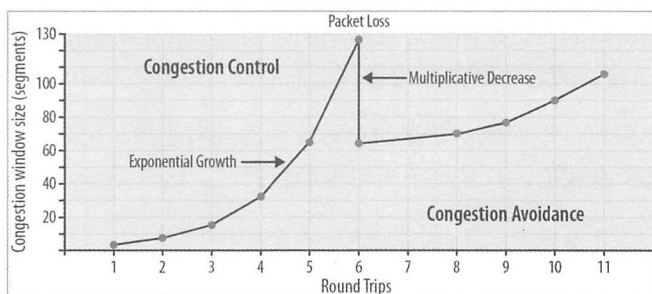


图 5-2

快速重传算法改进：等待超时定时器溢出会造成重传数据包的时间过长，通常在数秒，因此为了避免这个问题，TCP 专家们发明了快速重传算法，即如果发现 3 个重复的 ACK 确认，那么立即发起重传，从而减少网络耗时。

5.1.4 传统 TCP 拥塞控制问题

- 问题一：慢启动会造成网络耗时变长，拥塞窗口默认较小，无论多大的带宽，默认初始值都为 3。
- 问题二：当偶尔出现超时，发送报文段的个数会出现指数退避，直接减半，在某些情况下会导致网络利用率不足。
- 问题三：TCP 建立连接需要三次握手，但是如果握手期间发生超时或者丢包，只能等待超时定时器溢出，一般需要 3s，这会导致连接耗时过长。

5.2 Linux 内核升级中的 TCP 优化技术

Linux 集成了很多 Google 提出的 TCP 优化技术，从应用层面来说，了解这些技术可以在进行优化时有据可依，底层的优化在某些情况下，可以成为应用层优化非常重要的补充。从传统的 FEO 优化、用户流程优化、业务优化到内核优化，是格局和思想的转变和扩充，打开了视野，在 AliExpress 全球化用户体验提升过程中，Linux TCP 优化已经作为非常重要的手段来解决丢包引起的超时问题，并且取得了非常好的效果。尤其是一些物理距离特别远的国家，其用户在访问 AliExpress 网站时有着非常高的丢包率，所以有非常多的网络延迟超大的情况。

从 TCP 优化近几年的发展来看，Google 确实贡献了不少非常开放（Open）的想法，而且这些想法对业界整体技术的提升起到了至关重要的作用。特别是初始拥塞窗口的调整研究论





文，立足点虽小，但是效果非常明显。我们只有紧紧跟随技术潮流，了解业界走向，才能走到浪潮之巅，给客户带来不一样的体验，从而通过技术层面的优化来促进业务发展。

5.2.1 调整接收窗口

我们平时经常会遇到这种情况，租了 100MB 电信的宽带，但是下载速度并没有提升多少，还是每秒几兆字节的速度。这里面有多种原因，例如下载服务器的下行带宽有一定的上限，在高并发的同时进行下载，可能会造成带宽增至上限；还有中间路由器的缓冲区大小受限，以及运营商内部的某些限制，都会造成理论带宽和实际的下载速度不成比例，除这些因素，主要原因是接收窗口的 `rwnd` 设置不合理。

实际上接收窗口 `rwnd` 的合理值取决于 BDP 的大小，也就是带宽和延迟的乘积。假设带宽是 100Mb/s，延迟是 100ms，那么计算过程如下：

$$\text{BDP} = 100\text{Mb/s} \times 100\text{ms} = (100 / 8) \text{ MB/s} \times (100 / 1000) \text{ s} = 1.25\text{MB}$$

如果想最大限度提升 QPS，接收窗口 `rwnd` 不应小于 1.25MB。说点引申的内容：TCP 使用 16 位来记录窗口大小，也就是说最大值是 64KB，如果超过它，就需要使用 `TCP_Window_Scaling` 机制。

此优化方法在很多时候效果并不是很明显，可以尝试一下，因为实际上网络访问速度取决于多种条件，比如用户和机房的距离、中间路由器的拥塞状况，又如前面描述的能够发送报文段的个数由接收端窗口和拥塞窗口共同决定，而经过多少路由器是不能控制的，当然也可以和运营商谈，优化路由，但是成本可能会很大。

5.2.2 初始拥塞窗口调整（Linux 2.6.38 开始支持）

初始拥塞窗口的调整，可使 TCP 的传输效率得到极大提升，这是目前最好的方法。TCP 可以调整的参数达百项，大部分默认选项是适用于大部分情况的。Google 有篇论文完整地论证了拥塞窗口调大传输效率可提升 30% 左右，网络延迟可以减少约 30%。

大家都知道 HTTP 报文会按照 MSS 大小（一般为 1.46KB）分成多个 TCP 报文段进行传输，TCP 连接建立后，首次可以并行连续发送报文段的个数是由初始拥塞窗口大小决定的，对于 14.6KB 大小的 HTTP 报文，Linux 早期版本设置的初始拥塞窗口大小是 3，至少需要 4 个 RTT 才能完成传输。如果初始拥塞窗口是 10，那么只需要一个 RTT 即可发送完毕。这个窗口大小也适用于无线 TCP 优化的场景。



Linux 2.6.38 跳跃性地将初始拥塞窗口和初始接收窗口（Initial Receive Window）从 3 升到 10，这是非常简单而有效的方式。

初始拥塞窗口的调整是 Google 的工程师提出的，如图 5-3 所示。目前已经被业界广为接受，背景如下：

- 全球带宽提升明显。
- 对象个数在增加。
- 一旦丢包进入慢启动阶段，在很多情况下会通过调整初始拥塞窗口大小来进行传输。

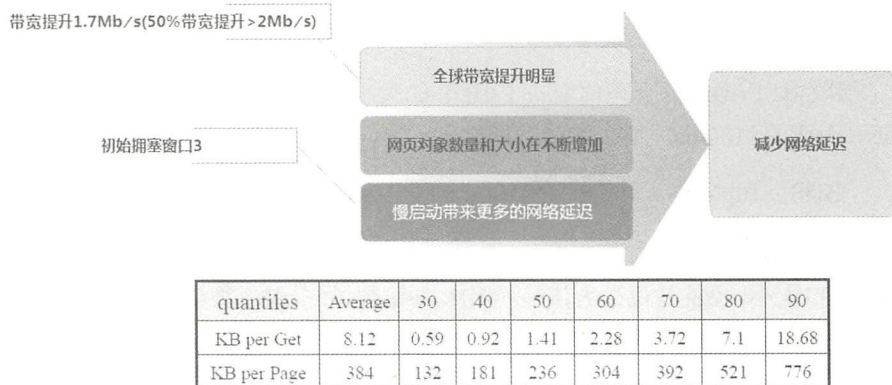


图 5-3

5.2.3 Early Retransmit（Linux 3.5 开始支持）

1. 快速重传的问题

快速重传通过 3 次重复 ACK，根据 ACK 报文中 sack 选项的内容，快速发现丢了哪个包，而不用等待超时定时器溢出时才发现，从而减少网络延迟。这个延迟不是由网络转发造成的，而是由 TCP 的重传机制造成的。

在下列情况下，快速重传不能发挥作用。

（1）当 TCP 拥塞窗口大小小于 4 时，如果第一个报文段丢失，由于没有足够的重复 ACK，造成 ACK 不足 3 个，快速重传将不能起作用。在 Linux 低版本下，初始拥塞窗口默认为 3，在 TCP 三次握手之后，连续能发送 3 个 TCP 报文段，当第一个报文段丢失，100% 只能靠重传定时器溢出来进行重传。

（2）ACK 丢失，在上面的情况下，窗口不够大（例如 4 个），也会造成 ACK 不足。



(3) 当 TCP 报文段在窗口结尾处丢包时（没有足够可用的报文段），由于本身没有后续的报文段发送，造成 ACK 不足。

2. Early Retransmit (ER) 的解决方案

TCP Early Retransmit 是 Google 提出的针对上述 ACK 不足情况的解决方案，Linux 3.5 将其作为 patch 引入默认的实现中，详情参见 <http://tools.ietf.org/html/rfc5827>。ER 是为了解决在窗口比较小的情况下，ACK 不足引起快速重传算法失效的问题。上述的第(3)种情况，可以采用淘宝内核团队的专家天澜提出的 Probe 算法解决，也可以使用 TLP 机制解决。

在 Linux 3.5 中，ER 选项默认是开启的，满足什么条件会触发 ER 呢？请看下面的源代码。

```
if (tp->do_early_retrans && ! retrans_out && tp->sacked_out &&
    (tp->packets_out == (tp->sacked_out + 1) && tp->packets_out < 4) &&
    ! TCP_may_send_now(sk))
    return 1;
```

请看图 5-4。

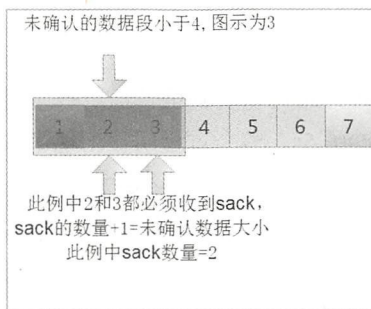


图 5-4

- `tp->do_early_retrans > 0`，启用 ER。
- `tp->retrans_out == 0`，没有重传且未确认的数据段。即当前窗口没有确认已经重传的报文段。
- `tp->sacked_out > 0`，有收到被 sack 的数据段。即有收到被 sack 的报文段，可以借此猜测哪个报文段丢失。
- `tp->packets_out < 4`，网络中发送且未确认的数据段小于 4 个。ACK 确认时，会将 sack 作为选项放在 ACK 中。在此例中，ACK 的确认序列号为 0，图示中窗口大小为 3，2 和 3 报文段的 sack 都属于 ACK 0（作为选项附加上去），因此图示的 3 个数据段从 ACK 的层面来说都属于已发送、未确认的情况。



- `tp->packets_out == tp->sacked_out + 1`, 被 sack 的数据段只比网络中发送且未确认的数据段少一个。即窗口中的没有丢失的报文段都必须收到 sack, 否则不会启用 ER。
- `TCP_may_send_now()` 为假, 此时不能发送新的数据段。如果延迟发送, 则不能启用 ER。

5.2.4 初始 RTO 调整 (Linux 2.6.18 开始支持)

在 TCP 三次握手过程中 (SYN、SYN+ACK、ACK), 如果出现 SYN 报文丢失, 第一次 RTO 无法度量, 会取 Linux 默认的 RTO, 作为重传定时器的超时时间。在跨境全球化的背景下, 由于地理距离长, 再加上 BGP 选路的复杂性 (出于成本考量, 运营商可能会采取限速, 也可能采取网络路由绕路的方式降低成本), 非常容易丢包。在连接开始的时候, 快速重传算法并不能起到作用, 只能根据重传定时器溢出来判断是否丢包, 所以默认 RTO 为 3s, 在跨境网络交互中, 会出现非常大的网络延迟 (3s 以上)。尽管大部分静态资源都采用 CDN 进行加速, 但是动态部分的请求仍然会打到源站上。例如, 用户浏览 AliExpress 的 detail 页面, 首先要从源站 (美国机房) 获取 detail 的 HTML 文档内容, 浏览器解析到 HTML 的内容, 才能下载静态图片、JS、CSS 等资源, 所以获取动态页面的 HTML 需要的时间对于整个页面的加载时间、首屏时间有非常大的影响。

从 Gomez 的 Last Mile 监控经常可以看到俄罗斯用户访问美国部署的机房, 仅获取 detail 页面的 HTML 文档的时间就超过了 5s, 整个过程明显出现了 SYN 报文的丢失, 只有等到重传定时器溢出才能进行 SYN 报文的重传, 所以 Google 提出的将 Linux 默认的 RTO 从 3s 降到 1s 的做法, 可以大大减少网络延迟, 对于供全球化用户访问的网站而言, 这个优化效果非常明显。

所以初始 RTO 调整非常适合解决跨境全球化背景下的网络延迟问题。但是初始 RTO 调整可能会带来一个问题, 当用户用手机访问时, 由于无线网络本身的不稳定和条件差等, 这个调整可能会造成网络更加拥塞, 所以在手机端调整 RTO 参数时需要特别注意。

5.2.5 TFO

TFO 是 TCP Fast Open 的简称, 传统的 TCP 需要经过三次握手才能传送数据, 而 TFO 在第三次 SYN 报文发送的时候会把请求报文段也随着 SYN 报文发送给接收端。Google 研究发现, TCP 三次握手是页面延迟时间的重要组成部分, 所以他们提出了 TFO: 在 TCP 握手期间交换数据, 这样可以减少一次 RTT。根据测试数据, TFO 可以减少 15% 的 HTTP 传输延迟, 全页面的下载时间平均节省 10%, 最高可达 40%。

第 1 步, 用户向服务器发送 SYN 包并请求 TFO Cookie。



第 2 步，服务器根据用户的 IP 地址加密生成 Cookie，随 SYN-ACK 发给用户。

第 3 步，用户缓存 TFO Cookie，并向服务器发送 SYN 包并携带 TCP Cookie，同时请求实体数据。

第 4 步，服务器校验 Cookie。如果合法，向用户发送 SYN+ACK，在用户回复 ACK 之前，便可以向用户传输数据；如果 Cookie 校验失败，则丢弃此 TFO 请求，视为一次普通 SYN，完成正常的三次握手。

目前 TFO 被植入了 Linux 2.6.34 内核，但是并没有被默认开启，在 Linux 3.13 内核中默认被开启。由于 TCP 是双工协议，客户端必须能够支持 TFO 的协议才能完成 TFO 的过程。Chrome 浏览器支持 TFO，但是在 Linux、Chrome 和 Android 操作系统中，TFO 默认是关闭的。

5.2.6 TSO

从严格意义上来说，TSO（TCP Segment Offload）和 GSO（Generic Segmentation Offload）并不是优化 TCP 传输效率的方式，而是网卡层面的优化，其目的是减少内核的 CPU 消耗。传统的 TCP 报文分段是在内核中完成的，而 TSO 和 GSO 将报文分片工作留给网卡来做，以减少内核的 CPU 消耗。

TCP 分段卸载在网络系统中，发送 TCP 数据之前，CPU 需要根据 MTU（一般为 1500）来将数据放到多个包中发送，对每个数据包都要添加 IP 头、TCP 头，分别计算 IP 校验和、TCP 校验和。如果有了支持 TSO 的网卡，CPU 可以直接将要发送的大数据发送到网卡上，由网卡硬件负责分片和计算校验和。TSO 带来一个不是问题的问题，即在计算重传率的时候，会引起重传率比实际的大。

$$\text{重传率} = \text{单位时间内重传的报文段} / \text{单位时间内已经传送出去的报文段}$$

这个计算是在内核层面进行的，而 TSO 在网卡层面进行分段，相当于分母变小了，自然 TSO 计算出来的结果会将重传率变大。

图 5-5 展示的是某个服务器重传率特别高的问题，用命令 `ethtool -k eth0` 可以查看 TSO 选项是否打开。

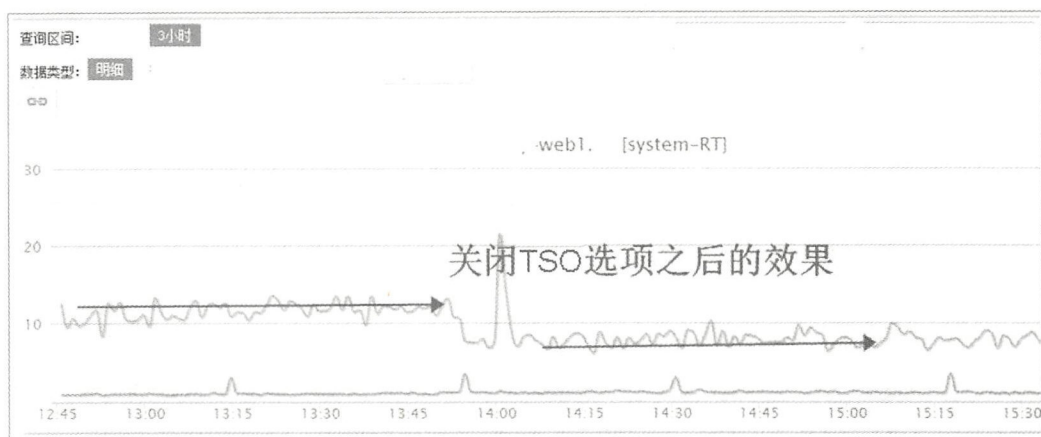


图 5-5

5.3 TIME_WAIT 问题案例分析

实际系统在线上运行时，经常在高并发时大量的 Socket 处于 TIME_WAIT 状态，导致 TCP 连接不可用。

5.3.1 问题现象

线上压力测试时出现大量的错误：cannot assign requested address: proxy: HTTP: attempt to connect to……QPS 达到 500 时就出现大量这样的错误。

压力测试的用例非常简单：施压机发送一个 URL 到 Apache，Apache 经过 Rewrite 先代理到 80 端口，再进行 Rewrite 并通过 Mod_JK 转发给 JBoss 进行处理，如图 5-6 所示。

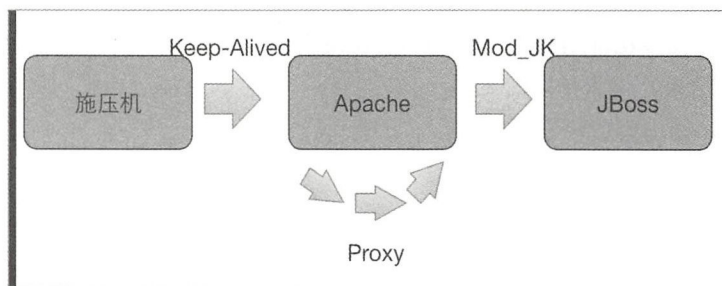


图 5-6

5.3.2 问题分析

从 5.3.1 节的错误提示可以看出，施压机和 Apache 服务器连接存在问题，首先确认一下连接数是否达到上限。

通过 `netstat` 统计命令查看 TCP 连接的使用情况，从下面的情况可以看出存在大量的 `TIME_WAIT` 状态连接，`TIME_WAIT` 一般由主动关闭端产生，可以确定是短连接。

```
netstat -ant|fgrep "："|cut -b 77-90 |sort |uniq -c
TIME_WAIT: 28000
Established 100
CLOSED_WAIT 500
```

简单描述一下 TCP 连接关闭的 4 次挥手。TCP 运用了可靠连接关闭，即经过双方的确认后，再关闭连接，避免双方因不知道连接关闭造成业务问题。如图 5-7 所示，如果客户端主动关闭连接，客户端先发送 FIN 包给服务器端，服务器端收到 FIN 包，回应 ACK 给客户端，表示服务器端已经准备关闭连接了，此时服务器端将连接状态设置为 `CLOSED_WAIT`，为了确认回应给客户端的 ACK 已经收到，服务器端再发送一个 FIN 包，服务器端的连接此时处于 `LAST_ACK` 状态，客户端接收到服务器端发送的 FIN 包之后，将连接状态设置为 `TIME_WAIT`，`TIME_WAIT` 状态存在的主要目的是防止迷途报文重现，影响新连接，所以要等待足够长的时间。特别是在公网传输过程中，有时发送报文后，如果接收端一直没有收到，那么可能会出现 ACK 不能回应给发送端的情况。客户端将 FIN ACK 回应给服务器端，服务器端收到后，将连接设置为 `CLOSED` 状态，此时将不再接收客户端发送的任何报文。客户端经过 2 个 MSL 大约 60s 将连接也设置为 `CLOSED` 状态。

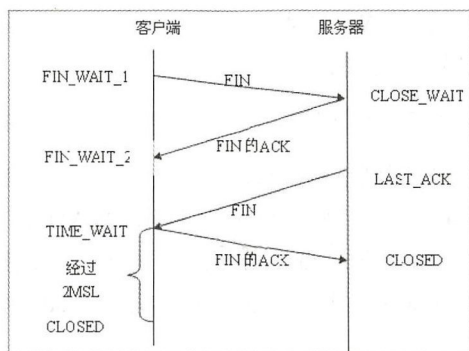


图 5-7

注意 TCP 是端到端的协议，如果连接不成功，可能是由于端口数量到达上限。查看服务器



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

的端口范围可以使用如下命令：

```
cat /proc/sys/net/ipv4/ip_local_port_range 32154 61500
```

本地启用的端口数量最多是 28000，也就是服务器能够建立的 TCP 连接数不能超过这个数量，可以看出本案例中端口的数量已经达到上限。

5.3.3 问题初步解决

针对这种情况，很明显要在 TIME_WAIT 上下工夫，要么减少 TIME_WAIT 状态的连接数量，要么定位到短连接的原因。在紧急情况下，如果对业务不清楚，而线上已经发生故障，需要在最短的时间内解决问题。

开启 TIME_WAIT 状态的 TCP 连接重用，可以看到 TIME_WAIT 状态的连接数量大幅下降，TIME_WAIT 状态的连接数量取决于连接的使用情况：

```
vi /etc/sysctl.conf
```

编辑：

```
net.ipv4.TCP_tw_reuse = 1 net.ipv4.TCP_tw_recycle = 1
```

运行生效：

```
/sbin/sysctl -p
```

5.3.4 问题再分析

刚才的焦点是通过降低 TIME_WAIT 状态的连接数量来减少对端口的占用，现在再回到短连接的问题上，Apache 的跳转规则如下：

```
RewriteRule ^/e/(.*)$ http://localhost:80/getKey.htm?token=$1 [L,QSA,P]
```

P 表示代理模式，是本地发起到本地的代理，通过建立 TCP 连接进行通信。有没有可能通过走本机的方式（不走 TCP 连接），更大幅度地减少 TIME_WAIT 状态连接的数量呢？从 Apache 官网发现，其实可以将 Proxy[P]修改成 pass through to next handler[PT]。P 模式是强制代理网络连接，而 PT 模式则只是做路径转换。这样修改之后就会走 Mod_JK，不需要采用本地连接，代码如下：

```
RewriteRule ^/e/(.*)$ http://localhost:80/getKey.htm?token=$1 [L,QSA,PT]
```

这样 TIME_WAIT 状态连接的数量基本降到 0，Apache 和 JBoss 通过长连接的方式进行通



信，因而不会造成 TCP 的主动关闭。

5.3.5 问题后记

一般 TIME_WAIT 状态连接容易过多，虽然这个状态对资源的消耗并不大，但是这种连接过多会造成端口数到达上限，这往往是由处理方式的不合理引起的。修改重用 TCP 连接，可能会造成迷途报文影响后续的请求，造成报文被破坏，一般在内网服务器之间可以如此处理，如果在公网上如此处理，迷途报文重现的可能性会大很多，所以任何问题都需要寻根究底找到根本原因。

5.4 总结

- 尽量将操作系统进行版本升级。高版本的 Linux 操作系统对 TCP 内核进行了多项优化，包括拥塞窗口变大、Early Transmit 等算法优化。
- 尽量通过监控工具发现问题，例如 LastMile。远距离的传输容易丢包，可以通过 LastMile 来发现问题。在建立连接的时候丢包，为了避免定时器溢出再进行报文的传输，可以通过调整初始 RTO 来减少网络耗时。
- 用户体验要注意 RTO。服务器端耗时占用整体耗时的一小部分，要把主要精力放在网络耗时的减少上。如果是小型网站，服务的用户群相对比较集中，就不要考虑 TCP 优化，因为带来的收益往往很小。
- 尽量减小 HTTP 报文的大小。因为 HTTP 报文越小，传输所需要的 RTT 次数越少，耗时越短，通常 Ajax 异步化是减少 HTTP 同步报文大小的主要手段。
- 尽量减少 HTTP 头的大小。HTTP 的 Gzip 压缩只针对 HTTP 主体，而 HTTP 头是不能被压缩的，要尽早对 Cookie 大小进行控制，因为 Cookie 往往是造成 HTTP 头偏大的主要因素，Cookie 大小在很多时候都接近 4KB 的上限。
- 尽量将机房部署在离核心用户近的地方，通过减少网络距离来减少网络耗时。
- 尽量选择大的运营商。小的运营商往往在 BGP 选路时会绕路，造成网络耗时增加。



6

第 6 章

DNS 优化

在大型网站构建的过程中，会遇到各种各样的问题。在人们的预想中，因为缓存的存在，TTL 内的 DNS 查询就在浏览器本地或者操作系统本地，不会向远程的域名服务器进行 DNS Lookup。但是在实际构建过程中，如果 DNS 架构不合理，会带来极差的用户体验。DNS 对于用户体验的影响如下：

- 对于页面请求域名，如果 DNS Lookup 时间过长会引起白屏时间过长。
- 对于图片对应的域名，如果 DNS Lookup 时间过长，白图的时间也会过长。

所有 DNS 一旦出现问题，对用户体验的影响是非常大的。前端性能优化部分已经讲述了如何利用浏览器的 DNS 预查询功能来解决 DNS 的性能问题，本章主要从 DNS 的架构方面来讲解 DNS 的优化。本章侧重优化，不是想把 DNS 的每个技术细节讲给读者，了解 DNS 的基本架构，对于 DNS 的优化和排查性能问题将起到关键的作用。另外就性能优化技术体系来说，由于 CDN 的调度和 DNS 的解析是密切相关的，所以对 DNS 的了解和熟悉是做 CDN 优化不可或缺的。



6.1 DNS 基本原理

6.1.1 DNS 的一些关键术语

我们经常会遇到各种专业术语，理解这些专业术语对高效沟通会起到一定的作用，在达成一致上比较有好处。

1. 根域名服务器（简称根域）

根域名服务器（Root Name Server）是互联网域名解析系统（DNS）中最高级别的域名服务器，负责返回顶级域名的权威域名服务器的地址。全球共有 13 个根域名服务器，共 504 个域名服务器（NameServer，NS），分散在世界各地，大部分采用 Anycast 技术，对外宣告同一个 IP 地址。在 DNS 查询过程中，由路由层面（BGP）寻找到具体承担解析任务的 NS 进行就近解析，以便提高解析效率。根域名服务器存放了顶级域的 NS 列表。

2. 顶级域名服务器（简称顶级域）

通常顶级域名指的是 .com（商业机构）、.net（网络提供商）、.edu（教育机构）、.cn（中国域名），顶级域名服务器是互联网域名解析系统中次高级别的域名服务器，负责返回权威域名服务器的地址。同样，全球共有 13 个 .com（或者其他几个）顶级域名服务器。

3. 权威域名服务器（简称权威域）

一般是指能够提供最终 IP 地址解析能力（最终 IP 地址是由这个 NS 来解析的）的一组服务器，或者能够控制最终 IP 地址解析结果的域名服务器（由权威 CNAME 给其他的 NS 来解析）。

4. Local DNS

用户上网需要通过 ISP 的网络接入互联网，ISP 会分配给用户一个 DNS 服务器，该服务器被称为 Local DNS，这个 DNS 代理域名解析请求给最终的权威 DNS，它具有 Cache 的能力，如果 Local DNS 在 TTL 时间内有 Cache，不会迭代向根域、顶级域和权威域发送 DNS 查询请求。

5. DNS no glue

在顶级域名授权中心注册一个新的域名时，需要提供（权威）NS 的列表，通常是一组域名。如果只是提供 NS 的域名，而没有 NS 的 IP 地址，那么在 DNS 解析过程中，需要解析 NS 的域名，并最终得到 NS 的 IP 地址，这样 UDP 查询才能知道把域名解析请求发送给谁。这种只是提供域名、没有提供 IP 地址的方式称为 no glue。



6. DNS TTL

简单地说，TTL（Time-To-Live）表示一条域名解析记录在 DNS 服务器上缓存的时间。当各地的 DNS 服务器接收到解析请求时，就会向域名指定的 DNS 服务器发出解析请求从而获得解析记录。在获得这个记录后，记录会在 DNS 服务器中保存一段时间，这段时间内如果再接到这个域名的解析请求，DNS 服务器将不再向域名指定的 DNS 服务器发出请求，而是直接返回刚才获得的记录，这个记录在 DNS 服务器上保留的时间就是 TTL 值。

注：与 IP 中的 TTL 不同，TTL 是 IP 协议包中的一个值，用于指定数据报被路由器丢弃前允许通过的网段数量。

6.1.2 DNS 查询过程

实际上 DNS 查询过程还是有点复杂的，不是普通小型网站的 3 层迭代，从根域到顶级域再到权威域。以解析 `www.aaa.com` 为例（`aaa` 为混淆后的名字），由于网站自身的需求（例如双机房的引流和负载均衡），会导致 DNS 架构变得复杂，如图 6-1 所示（图中对敏感内容进行了处理）。

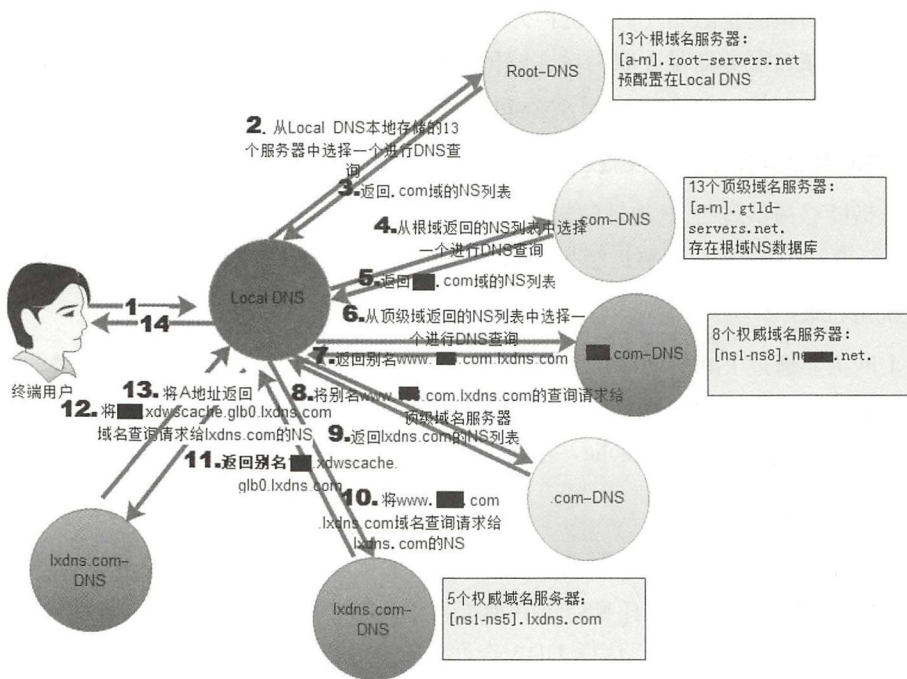


图 6-1



从图 6-1 中可以看出，实际 DNS 查询过程是比较复杂的，解析 `www.aaa.com` 需要经过 14 个步骤才能完成。

(1) 终端用户的浏览器发起 DNS 请求，发送到 ISP 提供给用户的 DNS 服务器（相当于由代理的 DNS 解析服务器迭代权威服务器返回的应答，然后将最终的 IP 地址返回给客户端浏览器）。

(2) ISP 的 DNS 服务器收到域名解析请求，将域名解析请求发送给其选定的根域名服务器（总共 13 个，由于根域名的 13 个地址几乎不发生变化，所以在 Local DNS 的软件配置文件中内置了 13 个根域名）。

注：一般而言 Local DNS 就是 ISP 提供的 DNS 服务器，但是严格地讲应该是用户终端设备上实际设置的 DNS，例如，可以指定 Google 的公共 DNS 8.8.8.8。

(3) 根域名服务器收到域名解析请求，发现是来源于 `.com` 顶级域名的请求，于是将 `.com` 顶级域名服务器列表和 IP 地址返回给 Local DNS。

(4) Local DNS 根据一定的策略选定 `.com` 的一个 NS，并且将 `www.xxx.com` 的域名查询请求发给这个 NS。

(5) 选定的 `.com` 的 NS 接收到 `www.aaa.com` 的域名查询请求，把在它这里注册和登记的 `aaa.com` 的 NS 列表和 IP 地址返回给 Local DNS。

(6) Local DNS 根据一定的策略选定 `aaa.com` 的一个 NS，并且将 `www.aaa.com` 的域名查询请求发给这个 NS。

(7) 选定的 `aaa.com` 的 NS 接收到 `www.aaa.com` 的域名查询请求后，将 `www.aaa.com` 的别名 `www.aaa.com.lxdns.com` 返回给 Local DNS（下面的任务就是解析这个域名）。

注：所谓 CNAME，就是将一个域名的解析请求转化成另外一个域名的解析请求（也可以称为委托，通常用于将网站性能要求委托给 CDN 提供商进行解析），另外一个域名请求的结果就是开始要解析域名请求所要的内容。例如，虽然发起的是 `www.aaa.com` 的域名解析请求，但是最终被域名解析服务器转化成 `www.aaa.com.lxdns.com` 的解析请求，所以经过 CNAME 之后，就解析 CNAME 的那个域名了。

(8) 要解析 `www.aaa.com.lxdns.com` 仍然要有完整的迭代过程。首先还是要把解析请求发送给根域名服务器（根域名服务器的 TTL 通常为几天），然后 Local DNS 在本地的服务器列表缓存中选择一个顶级域名服务器，并且将请求发送给该服务器。



(9) 选定的.com 的 NS 接收到 `www.aaa.com.lxdns.com` 的域名查询请求后，将 NS 列表和 IP 地址返回给 `lxdns.com`。

(10) 将 `www.163.com.lxdns.com` 的域名解析请求发送给 Local DNS 选定的 NS。

(11) `lxdns.com` 的 NS 接收到 `www.aaa.com.lxdns.com` 的域名解析请求后，将别名 `aaa.xdwscache.glb0.lxdns.com` 返回给 Local DNS CNAME。

(12) 要解析 `aaa.xdwscache.glb0.lxdns.com` 这个域名，首先要知道 `lxdns.com` 的 NS 地址。因为之前访问过，所以 Local DNS 有缓存，那么就将这个解析请求发送给选定的其中一个 `lxdns.com` 的域名解析服务器。

(13) 选定的 `lxdns.com` 的域名解析服务器接收到域名解析请求后，将 A 地址返回给 Local DNS。

(14) Local DNS 再将收到的 A 地址返回给客户端，并且将其保存在高速缓存中。

可以看出，DNS 的架构如果不合理，对高性能用户体验是有影响的，如果每个步骤需要 20ms，这 14 个步骤加起来是 280ms 的响应时间，对于一个图片的加载来说是非常慢的。

6.1.3 NS 选择策略和机制

从上面的 DNS 查询过程来看，有很多关于 Local DNS 如何选择 NS 的描述，那么 Local DNS 如何从一堆 NS 列表中选择一个它认为比较好的呢？如果选择不好，对于性能有很大的影响，如果只是轮询，则 DNS 的优化会比较难做。为了可用性和性能，各个主流的 DNS 是如何做选择的？我们无法控制 Local DNS，但是知道它们是如何进行选择的，至少可以帮助我们做技术方案和策略。

目前 DNS 配置中 BIND 是主流，占有率最高。下面比较一下 BIND(9.7.3 和 9.8.0)、PowerDNS(3.1.5)、Unbound(1.4.10)、DNS Cache(1.05 版本)和 Windows DNS 6.1(Windows Server 2008)。

我们来看一下主流的 DNS 软件的 NS 选择策略是什么。UCLA 实验室的研究人员经过大量的样本测试，得出以下结论。

1. 在 NS 负载正常的情况下，DNS 如何选择

1) 部分 DNS 软件优先选择最快的 NS

- PowerDNS：总是选择网络延迟最少的 NS。
- BIND：网络延迟最少的 NS 被选中的概率大，但是每个 NS 都有一定的概率被选中。



2) 部分 DNS 软件没有将 RTT 作为选择因子

- DNSCache: 不会测量 RTT, 对所有的 NS 进行随机选择。
- Unbound: 会测量 RTT, 选择 $SRTT < 400ms$ 的 NS。

3) 部分 DNS 软件将解析请求发给 RTT 大的服务器

BIND 总是给 RTT 最长的 NS 以更大的选择概率, 可能存在 Bug。

2. DNS 软件如何处理没有响应的 NS

- BIND 可能会发送更多的请求给没有响应的 NS。
- 从测试结果来看, Windows DNS 不会选择没有响应的 NS。
- Unbound 会检测到没有响应的 NS, 不会发送解析请求给没有响应的 NS。
- PowerDNS 总是会选择 RTT 短的 NS。
- DNSCache 会均等选择 NS, 即使没有响应。

3. 关于 NS 选择策略的结论

- 主流的 DNS 软件有关 BIND 的新的设计, 已经朝向以 RTT 较短的 NS 为主。
- Local DNS 安装的软件, 有老的版本也有新的版本, RTT 长的 NS 仍然会被选择到。
- 没有响应的 NS 会严重影响 DNS Lookup, 所以 NS 的监控很关键。

4. NS 选择策略对于 DNS 优化的参考意义

大多数新版本的 DNS 软件已经支持小 RTT 选择 NS 的能力, 目前在跨洲、跨地区等远距离情况下, DNS 部署架构需要考虑多点就近部署。如何就近选择呢? 只要在 4 个 NS 域名下面挂一个新的离用户较近的 VIP, 就可以提升 DNS 的查找性能、减少网络延迟。也就是说, NS 的选择原理就是通过多点部署让 DNS 就近选择。

6.1.4 DNS 扩展协议 EDNS

EDNS 是 Extension Mechanisms for DNS 的缩写。在做精确调度时, EDNS 协议能起到非常关键的作用。在 CDN 调度过程中, 一般都根据 Local DNS 的 IP 地址进行静态的域调度, 这个协议导致了很多问题。

CDN 的调度存在调度不准的情况, 大都跟上面的机制相关。一般来说, CDN 的 IP 地址库和国家都是比较精确的, 因为 IP 地址和国家、地区的关系, 都在某个地方注册、登记过, 所以只要 IP 地址来源确定, 地区就可以确定。





但是当 Local DNS 的 IP 地址和用户所在地区的 IP 地址不在同一地区时，问题就出现了。例如，一个跨国企业，人在中国上班，但是机房的 Local DNS 可能在美国总部，CDN 调度时会把美国的 CDN 节点提供给中国上班的用户访问，此时会造成很大的性能问题。再比如，用户使用的 Local DNS 是 Google 的 8.8.8.8，这时就会出现调度问题。

为了解决这个问题，DNS 的扩展协议 EDNS 出炉了，EDNS 是 Google 提交的一份 DNS 扩展协议，允许 Local DNS 将用户的真实 IP 地址传递给 NS 或者 CDN 全局调度器，这样 CDN 的调度就精准了。但是协议都有双工性，不仅 Local DNS 需要遵守和实现该协议，还需要 CDN 对协议具有解析的能力，EDNS 协议目前还只是在公共的 DNS 上得到了很好的应用，其他地方仍然很难进行推广。

6.1.5 常用 DNS 相关命令

为了能够清楚地知道 DNS 的性能问题，必须了解一些常用的 DNS 相关命令。

1. 查看 DNS 的完整解析路径

Linux 命令：

```
dig +trace 域名 +additional
```

功能：不通过缓存获取 DNS 的完整解析路径，包括每个 NS 的域名和 IP 地址。可以通过查看解析路径来判断是否可以去除多余的 CNAME，每一层 CNAME 都要经过至少一次 RTT 损耗；查看 NS 的部署结构是否合理，这些 NS 的分布是否和终端用户的分布一致，是否就近部署了架构，是否存在超远距离 UDP 查询的情况。

例如，解析 www.aaa.com 的完整路径如下。

第 1 步，找到 www.aaa.com 的权威 DNS，并返回 CNAME 的别名 www.aaa.com.lxdns.com。

```
; <>> Dig 9.8.3-P1 <>> +trace www.163.com +additional
;; global options: +cmd
.          37181  IN  NS  a.root-servers.net.
.          .....
.          37181  IN  NS  m.root-servers.net.
a.root-servers.net. 3550800 IN  A   197.41.0.4
.....
h.root-servers.net. 3552076 IN  A   127.63.2.53
;; Received 496 bytes from 202.101.172.35#53(202.101.172.35) in 11 ms
com.          172800  IN  NS  f.gtld-servers.net.
.....
```





```
com.          172800 IN NS m.gtld-servers.net.
a.gtld-servers.net. 172800 IN A 193.5.6.30
.....
l.gtld-servers.net. 172800 IN A 193.41.162.30
;; Received 501 bytes from 192.58.128.30#53(192.58.128.30) in 52 ms
aaa.com.      172800 IN NS ns1.bbb.net.
.....
aaa.com.      172800 IN NS ns8.bbb.net.
ns1.bbb.net.  172800 IN A 123.58.174.177
.....
ns8.bbb.net.  172800 IN A 54.178.189.118
ns8.bbb.net.  172800 IN A 54.86.139.107
;; Received 342 bytes from 192.12.94.30#53(192.12.94.30) in 957 ms
www.aaa.com.  600 IN CNAME www.bbb.com.lxdns.com.
;; Received 61 bytes from 123.58.173.178#53(123.58.173.178) in 4 ms
```

可以看到, `dig` 只 `trace` 到第一层 `CNAME` 就终止了, 要查看完整的路径怎么办?

如前面所说 `CNAME` 是别名, 由一个域名查询任务变成另外一个域名的查询。因此 `CNAME` 只要继续别名的 `DNS` 查询就可以了。

第 2 步, 找到 `www.bbb.com.lxdns.com` 的权威 `DNS`, 返回另一个别名 `aaa.xdwscache.glb0.lxdns.com`, 并把这个域名的权威 `DNS` 直接返回给 `Local DNS`。

```
; <<>> DiG 9.8.3-P1 <<>> +trace www.bbb.com.lxdns.com +additional
;; global options: +cmd
.          68568 IN NS a.root-servers.net.
.....
.          68568 IN NS m.root-servers.net.
a.root-servers.net. 3510126 IN A 198.41.0.4
.....
h.root-servers.net. 3582189 IN A 128.63.2.53
;; Received 496 bytes from 202.101.172.35#53(202.101.172.35) in 19 ms

com.      172800 IN NS a.gtld-servers.net.
.....
com.      172800 IN NS m.gtld-servers.net.
a.gtld-servers.net. 172800 IN A 192.5.6.30
.....
k.gtld-servers.net. 172800 IN A 192.52.178.30
;; Received 511 bytes from 192.5.5.241#53(192.5.5.241) in 45 ms

lxdns.com. 172800 IN NS ns1.lxdns.com.
```





```
.....
lxdns.com.      172800 IN NS ns5.lxdns.com.
ns1.lxdns.com.   172800 IN A  113.107.57.68
.....
ns5.lxdns.com.   172800 IN A  115.231.158.58
;; Received 209 bytes from 192.5.6.30#53(192.5.6.30) in 323 ms

www.aaa.com.lxdns.com. 600 IN CNAME  aaa.xdwscache.glb0.lxdns.com.
glb0.lxdns.com.      172800 IN NS  n3.glb0.lxdns.com.
glb0.lxdns.com.      172800 IN NS  n4.glb0.lxdns.com.
glb0.lxdns.com.      172800 IN NS  n5.glb0.lxdns.com.
glb0.lxdns.com.      172800 IN NS  n1.glb0.lxdns.com.
glb0.lxdns.com.      172800 IN NS  n2.glb0.lxdns.com.
n3.glb0.lxdns.com.   172800 IN A  222.186.17.61
n4.glb0.lxdns.com.   172800 IN A  125.39.1.117
n5.glb0.lxdns.com.   172800 IN A  222.132.5.104
n1.glb0.lxdns.com.   172800 IN A  58.220.6.138
n2.glb0.lxdns.com.   172800 IN A  218.60.106.128
;; Received 237 bytes from 115.231.158.60#53(115.231.158.60) in 13 ms
```

第 3 步，Local DNS 从上一步返回 glb0.lxdns.com 的列表中选择一个 NS，并将 aaa.xdwscache.glb0.lxdns.com 的域名解析请求发送给这个 NS，n*.glb0.lxdns.com 返回给 xdwscache.glb0.lxdns.com 一个 NS 列表，aaa.xdwscache.glb0.lxdns.com 的域名解析请求再发送给其中的一个 NS，选择的 NS 返回给 Local DNS 并解析最终的 A 地址，由此最终的 IP 地址终于解析出来了。

```
; <<>> DiG 9.8.3-P1 <<>> +trace 163.xdwscache.glb0.lxdns.com +additional
.....//省略根域和顶级域的迭代
n1.glb0.lxdns.com. 172800 IN A  58.220.6.138
n2.glb0.lxdns.com. 172800 IN A  218.60.106.128
n3.glb0.lxdns.com. 172800 IN A  222.186.17.61
n4.glb0.lxdns.com. 172800 IN A  125.39.1.117
n5.glb0.lxdns.com. 172800 IN A  222.132.5.104
;; Received 211 bytes from 125.39.1.107#53(125.39.1.107) in 143 ms

aaa.xdwscache.glb0.lxdns.com. 120 IN  A  183.131.119.87
aaa.xdwscache.glb0.lxdns.com. 120 IN  A  183.131.119.86
xdwscache.glb0.lxdns.com. 172800 IN NS  n4.glb0.lxdns.com.
xdwscache.glb0.lxdns.com. 172800 IN NS  n1.glb0.lxdns.com.
xdwscache.glb0.lxdns.com. 172800 IN NS  n2.glb0.lxdns.com.
xdwscache.glb0.lxdns.com. 172800 IN NS  n5.glb0.lxdns.com.
xdwscache.glb0.lxdns.com. 172800 IN NS  n3.glb0.lxdns.com.
```





```
n1.glb0.lxdns.com. 7200 IN A 222.186.17.58
n2.glb0.lxdns.com. 7200 IN A 111.206.217.82
n3.glb0.lxdns.com. 7200 IN A 115.231.84.177
n4.glb0.lxdns.com. 7200 IN A 101.227.66.164
n5.glb0.lxdns.com. 7200 IN A 111.206.217.64
;; Received 248 bytes from 111.206.217.82#53(111.206.217.82) in 153 ms
```

aaa 站点的 DNS 解析层次多而复杂, 除非 NS 的部署足够合理, 否则性能很难得到保障。当然 DNS 设置这么多层次肯定有历史原因, DNS 问题分析的目的也是为了解决历史问题, 并将架构升级, 以使用户得到更好的性能体验。

2. 指定 Local DNS 进行域名解析

Linux 命令:

```
dig @LocalDNSIP 域名
```

功能: 指定 Local DNS 进行域名代理, 通常可以用于公共的 Local DNS, 如 Google 的 8.8.8.8, 因为在 CDN 上大都通过 Local DNS 的地址进行调度, 因此可以使用这个方法进行 CDN 的调度测试。

3. 查看域名对应的 NS

```
dig ccc.com ns (ccc 为混淆后的代称名)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2159
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;ccc.com. IN NS

;; ANSWER SECTION:
ccc.com. 3600 IN NS nsp.ccconline.com.
ccc.com. 3600 IN NS nshz.ccconline.com.
ccc.com. 3600 IN NS nsp2.ccconline.com.
ccc.com. 3600 IN NS ns8.ccconline.com.
```

4. 查看 DNS 的 TTL

用 dig 等命令查看 DNS Lookup 解析路径时, 会发现 TTL 值一直在变化, 要准确地查看某个域名对应的 TTL 初始值, 需要用下列命令。

```
dig @nsdomain/ip 域名
```





查看到对应的 NS 后，就可以用这个命令查看 TTL 到底是多少了：

```
dig @nsp.ccconline.com www.aaa.com

;www.aaa.com. IN A

;; ANSWER SECTION:
www.aaa.com. 600 IN A 41.156.190.86

;; AUTHORITY SECTION:
aaa.com. 3600 IN NS nssz.ccconline.com.
aaa.com. 3600 IN NS nsp.ccconline.com.
aaa.com. 3600 IN NS ns8.ccconline.com.
aaa.com. 3600 IN NS nsp2.ccconline.com.
```

5. 测量网络延迟

ping ServerIP

功能：从 ping 操作所在的客户端默认发送一个 32 字节的包给路由器来测量 RTT。通常 RTT 越大距离越远，可以大体测量网络延迟，以便选择合适的地点建立 DNS 解析点。

6.2 实战案例：超远距离 DNS 性能问题分析和优化

6.2.1 现象描述

从 Gomez 上 LastMile 监控点的抽样调查中发现，DNS Lookup 时间长，如图 6-2 和图 6-3 所示，DNS Lookup 的时间均在 1s 以上，多个时间超过 2s。

| Object Name | Type | IP Address | Return Code | Total (end-to-end) | DNS lookup time | Connection time | Sockets Layer Time | 1st byte download | Content download | Number of Bytes |
|------------------------------------|-------------------------|------------|-------------|--------------------|-----------------|-----------------|--------------------|-------------------|------------------|-----------------|
| DNS Lookup http://www.163.com | DNS Lookup | 119.2.1.1 | 0 | 21.897 | 1.021 | 0.005 | | | | 35726 |
| Connection http://www.163.com (C0) | Connection | 119.2.1.1 | 302 | 2.193 | | | | 2.193 <0.001 | | 0 |
| /ru_home.htm(C0) | text/html;charset=UTF-8 | 119.2.1.1 | 200 | 1.473 | | | | 1.473 <0.001 | | 35099 |
| DNS Lookup http://style.163.com | DNS Lookup | 119.2.1.1 | 0 | 1.159 | 1.77 | | | | | |
| DNS Lookup http://vk.com | DNS Lookup | 119.2.1.1 | 0 | 1.120 | 2.274 | | | | | |
| DNS Lookup http://olmat.com | DNS Lookup | 119.2.1.1 | 0 | 1.140 | 10.018 | | | | | |
| DNS Lookup http://03.163.com | DNS Lookup | 119.2.1.1 | 0 | 1.184 | 5.006 | | | | | |
| DNS Lookup http://02.163.com | DNS Lookup | 119.2.1.1 | 0 | 1.168 | 9.51 | | | | | |
| DNS Lookup http://img.163.com | DNS Lookup | 119.2.1.1 | 0 | 1.169 | 10.472 | | | | | |

DNS Lookup的时间
均为1s以上

图 6-2





| Object Name | Type | IP Address | Return Code | Total (end-to-end) | DNS lookup | Connection time | Sockets Layer Time | 1st byte | Content download | Number of Bytes |
|--|--------------------------|------------|-------------|--------------------|------------|-----------------|--------------------|----------|------------------|-----------------|
| DNS Lookup http://www.i.com | DNS Lookup | 6.1 | | | 7.014 | | | | | |
| Connection 0 | Connection | 6.1 | | 37.562 | | | | | | 81446 |
| http://www.i.com/charset=UTF-8 | text/html; charset=UTF-8 | 6.1 | 302 | 0.216 | | | | 0.216 | <0.001 | 0 |
| /v_home.htm(C0) | text/html; charset=UTF-8 | 6.1 | 200 | 1.919 | | | | 0.25 | 1.669 | 34254 |
| DNS Lookup http://img.i.com | DNS Lookup | 0.56 | | | 7.169 | | | | | |
| DNS Lookup http://style.i.com | DNS Lookup | 0.58 | | | 7.004 | | | | | |
| DNS Lookup http://vk.com | DNS Lookup | 1.118 | | | 0.217 | | | | | |
| DNS Lookup http://qtms03.i.com | DNS Lookup | 6.250 | | | 4.079 | | | | | |
| DNS Lookup http://03.i.com | DNS Lookup | 0.50 | | | 7.005 | | | | | |
| DNS Lookup http://02.i.com | DNS Lookup | 0.35 | | | 4.071 | | | | | |
| Connection 1 | Connection | 1.118 | | 7.837 | | 0.026 | | | | 26351 |
| /a/ap/openapi.js(C1) | application/x-javascript | 1.118 | 200 | 0.053 | | | | 0.027 | 0.026 | 20108 |
| DNS Lookup http://qtms02.i.com | DNS Lookup | 6.240 | | | 7.013 | | | | | |
| Connection 2 | Connection | 6.250 | | 0.033 | | 0.013 | | | | 3800 |
| Connection 3 | Connection | 0.35 | | 0.222 | | 0.023 | | | | 101265 |
| Connection 4 | Connection | 0.35 | | 15.358 | | 0.023 | | | | 118544 |
| Connection 5 | Connection | 0.35 | | 15.743 | | 0.023 | | | | 163238 |
| Connection 6 | Connection | 0.35 | | 0.229 | | 0.023 | | | | 92804 |
| Connection 7 | Connection | 0.35 | | 0.225 | | 0.023 | | | | 114923 |
| Connection 8 | Connection | 0.35 | | 0.231 | | 0.024 | | | | 90889 |
| /aps/0/T1uf91FrXXXarO1nv-150-100.jpg(C2) | image/jpeg | 6.250 | 200 | 0.019 | | | | 0.018 | 0.001 | 3800 |

CDN的CNAME动作同样耗时严重，有的甚至达到了7s

图 6-3

6.2.2 DNS Lookup 耗时长的问题分析

分析 DNS Lookup 耗时过长的问题，需要从 DNS 的部署架构和业务场景开始，请看图 6-4。

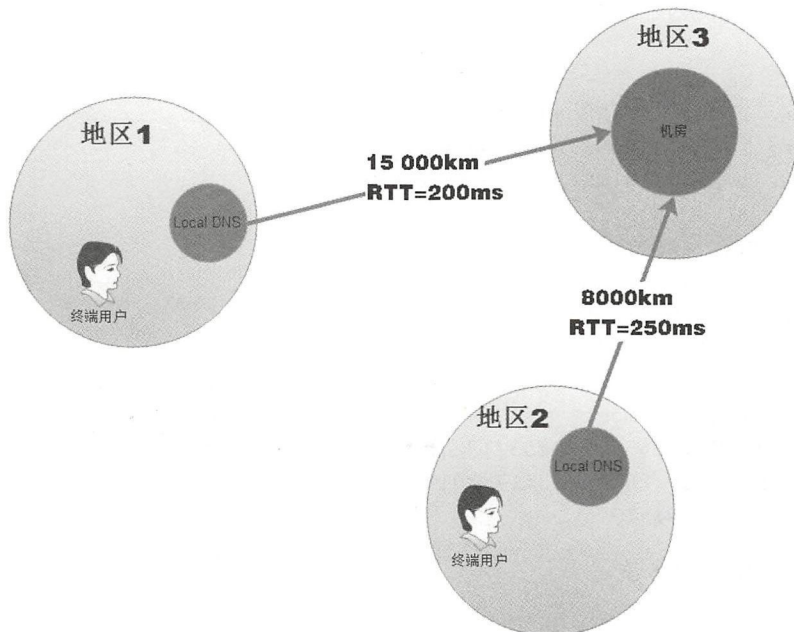


图 6-4





从图 6-4 中可以看出，终端用户到机房的物理距离远，访问网站的用户遍布全球。机房在地区 3，终端用户在地区 1、地区 2，网站的权威 DNS 部署在地区 3 的机房，地区 1 到机房的物理距离达到 15000km，网络延迟 200ms（RTT，使用 ping 测量）；地区 2 到机房的物理距离 8000km。地区 2 网络建设落后，在全球网速排名接近 80 位，网络延迟 250ms（RTT，使用 ping 测量）。

再以域名 gtm*.aaa.com 的解析为例，对解析层次和 NS 的部署分布进行分析。

(1)alicdn.com 的权威 DNS 部署在国内，而用户在国外，一次 UDP 查询可能跨越 20000km，一次完整的 RTT 超过 500ms，如果丢包则会会长达数秒。

```
$ dig ns alicdn.com

; <<>> DiG 9.3.4-P1 <<>> ns alicdn.com
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37821
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 3

;; QUESTION SECTION:
;acd.com. IN NS

;; ANSWER SECTION:
acd.com. 172800 IN NS ns5.aun.com.
acd.com. 172800 IN NS ns4.aun.com.
acd.com. 172800 IN NS ns3.aun.com.

;; ADDITIONAL SECTION:
ns3.aun.com. 180692 IN A 115.124.17.155
ns4.aun.com. 180692 IN A 110.75.20.27
ns5.aun.com. 180692 IN A 110.75.38.28

;; Query time: 13 msec
;; SERVER: 172.22.57.142#53(172.22.57.142)
;; WHEN: Mon May 26 14:23:17 2014
;; MSG SIZE rcvd: 137
```

(2) CNAME 达到 5 次。

```
gt01.acd.com. 3600 IN CNAME gtms01.alicdn.com.danuoyi.tbcache.com.
gt01.alicdn.com.danuoyi.tbcache.com. 1800 IN CNAME ptms01.danuoyi.tbcache.
```





```
com.  
ptms01.danuoyi.tbcache.com. 300 IN CNAME gtms.acdn.com.edgesuite.net.  
gtms.alicdn.com.edgesuite.net. 21600 IN CNAME a1049.g.akamai.net.  
a1049.g.akamai.net. 20 IN A 128.241.216.194  
a1049.g.akamai.net. 20 IN A 128.241.216.154
```

对这个部署架构而言，造成 DNS Lookup 时间长的可能原因如下。

- 可能原因一：DNS 部署结构不合理和解析层次设置不合理，造成了多次超远距离的 UDP 查询。物理距离过远，RTT 达到 200ms 以上，解析层次多，可能解析层次超出了预期，多次 CNAME DNS Lookup，经过 5 次 DNS 迭代就可以超过 1.5s。
- 可能原因二：DNS 设置的 TTL 时间过短，每次测试都需要到权威 DNS 去迭代，造成远距离交互，导致延迟。

6.2.3 DNS 解析性能解决方案

1. 设置合理的 TTL 时间

TTL 时间一旦过期，请求会返回到权威 DNS 上，超远距离提供服务的网站，用户分布在离机房位置超远距离的国家，这些国家实际上与权威 DNS 所在的地区机房距离很长，一旦请求发送到美国的权威 DNS 上，加上丢包因素会造成 DNS Lookup 时间非常长。众所周知，TTL 设置短的目的是为了高可用性，无论是 CDN 还是淘宝这样的网站的动态请求，设置很短的 TTL 时间，在一个地方出问题，可以通过简单的域名切换到另外一个地方，但是如果断定没有第二条路可走，TTL 设置过短实际上对性能有很大的损耗。

对于像淘宝这样规模网站的主域名，TTL 设置可以短一点，对于大型网站，异地容灾基本都是标配，一旦机房 A 出现问题，就将域名解析的 A 地址从地区 A 的虚拟 IP 切换到地区 B 的虚拟 IP，这样能够起到异地容灾的作用，大大提高系统的稳定性。所以对于笔者曾经所在业务的主域名，由于网站规模较大，TTL 时间并没有调整，保持 10 分钟不变。

对于静态资源相关的域名，跨境全球化网站的 CDN 解析路径比较长，这有客观原因，海外 CDN 目前可能是多个 CDN 厂商共存的状态，各 CDN 厂商在全球的部署节点数有所不同，服务优势有差异。首先要 CNAME 给 CDN 提供商 A 的解析服务器（其目的是根据国家来确定是否 CNAME 给该 CDN 厂商进行静态加速），再由 AliCDN 的解析服务器决定是否 CNAME 给 Akamai 的全局调度解析器。CNAME 的 TTL 时间设置过短会导致一次完整的 DNS 的递归查询，需要多个 RTT 往返，这对于跨境全球化网站的性能而言是一场灾难。

笔者所在跨境电商网站早期将 CDN 相关的第一层 CNAME 时间全部改成了 1 小时，DNS



Lookup 抖动小了很多。第一层 CNAME 的 TTL 改长，那么在设定的 TTL 内，gtms01.alicdn.com.danuoyi.tbcache.com 这一层的域名解析可以省掉，Local DNS 可以直接通过 Local DNS 缓存的 IP 地址访问 gtms01.alicdn.com.danuoyi.tbcache.com 的权威 DNS。但是从下面仍然可以看到第二层、第三层的 CNAME 的 TTL 时间还是有点短，个人觉得还可以调整至更长时间，特别是第三层，从解析层次上来看，DNS 完整的查询时间在最差的情况下可能需要数十个 RTT 往返，所以几秒钟就正常了。参考代码如下。

```
gtms01.alicdn.com.3600 IN CNAME gtms01.alicdn.com.danuoyi.tbcache.com.
gtms01.alicdn.com.danuoyi.tbcache.com.1800 IN CNAME piscestms01.danuoyi.
tbcache.com. piscestms01.danuoyi.tbcache.com. 300 IN CNAME gtms.alicdn.com.
edgesuite.net.gtms.alicdn.com.edgesuite.net.21600 IN CNAME a1049.g.akamai.
net. a1049.g.akamai.net.20 IN A 128.241.216.194 a1049.g.akamai.net.20 IN A
128.241.216.154
```

2. DNS 多点就近部署架构升级

TTL 时间的调整能够大幅减少用户所在的 Local DNS 到权威 DNS 的往返时间，但是 TTL 时间一旦过期，特别是有些层次的 TTL 时间无法变得更长时（如高可用性要求）。权威 DNS 的多点部署非常关键，域名解析原理看起来比较简单，但在实际使用过程中会有非常多的 CNAME，每一次 CNAME 都有一次完整的域名解析请求，需要数个 RTT 才能完成。gtms01.alicdn.com 域名的权威 DNS 开始只在国内有，为了优化 AliCDN 相关域名的 DNS 解析性能，笔者做了两件事情。

第一件事情是对权威 NS 的变更。把 alicdn.com 的权威 NS，从国内改成 AliExpress 主域名的 NS，变更之前如下：

```
alicdn.com. 172800 IN NS ns5.aliyun.com.
alicdn.com. 172800 IN NS ns4.aliyun.com.
alicdn.com. 172800 IN NS ns3.aliyun.com.
```

变更之后如下：

```
;; ANSWER SECTION:
alicdn.com. 172800 IN NS nshz.alibabaonline.com.
alicdn.com. 172800 IN NS nsp.alibabaonline.com.
alicdn.com. 172800 IN NS ns8.alibabaonline.com.
alicdn.com. 172800 IN NS nsp2.alibabaonline.com.
```

第二件事情是在 taocache.com（AliExpress 静态域名相关的调度解析服务器）的 NS 下面挂了一个新的美国的 IP 地址，大家知道 Local DNS 现在已经很智能了，可以就近选择一个 NS 地

址，这就是利用 DNS 的多点就近部署架构来解决 DNS 的性能问题，如图 6-5 和图 6-6 所示。

```
;; AUTHORITY SECTION:
danuoyi.tbcache.com. 287 IN NS danuoyins5.tbcache.com.
danuoyi.tbcache.com. 287 IN NS danuoyins6.tbcache.com.
danuoyi.tbcache.com. 287 IN NS danuoyins2.tbcache.com.
danuoyi.tbcache.com. 287 IN NS danuoyins3.tbcache.com.
danuoyi.tbcache.com. 287 IN NS danuoyins1.tbcache.com.
danuoyi.tbcache.com. 287 IN NS danuoyins4.tbcache.com.

;; ADDITIONAL SECTION:
danuoyins1.tbcache.com. 355 IN A 115.124.17.156
danuoyins2.tbcache.com. 355 IN A 110.75.20.28
danuoyins3.tbcache.com. 355 IN A 110.75.29.28
danuoyins4.tbcache.com. 355 IN A 115.28.122.200
danuoyins5.tbcache.com. 355 IN A 205.204.114.23
danuoyins6.tbcache.com. 426 IN A 115.28.122.201
```

图 6-5

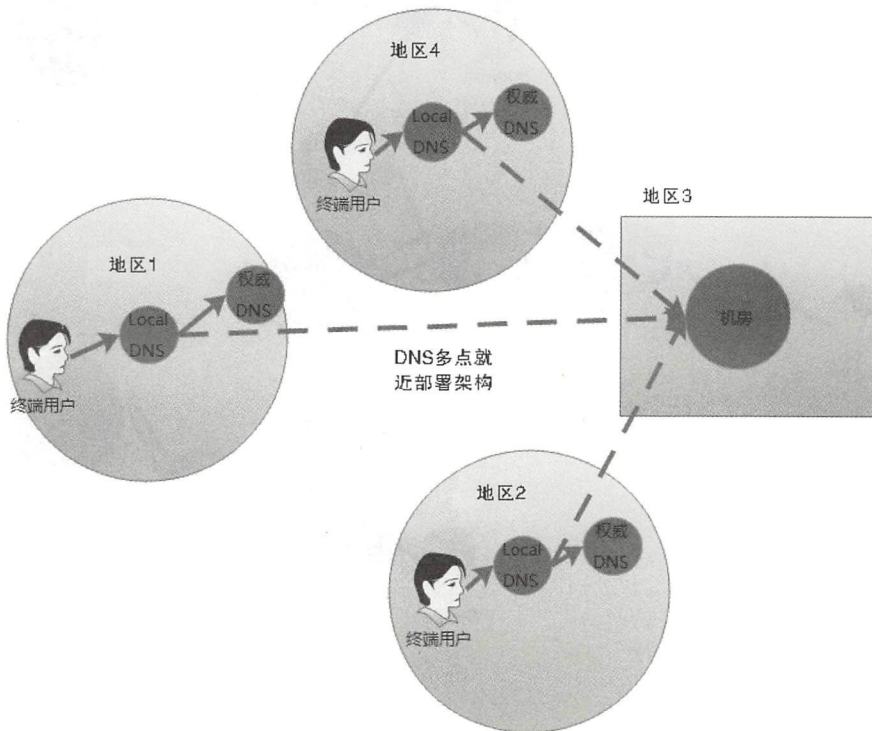


图 6-6

多点就近部署架构本质上是根据 Local DNS 的低延迟选中策略，在离用户近的地方部署 NS，Local DNS 从一堆 NS 的列表里就近选择一个来承担解析任务。



3. 基于 Anycast 的 DNS 解析架构——未来 DNS 优化方向

所谓 Anycast 的 DNS 架构，是多台 NS 宣告同一个 Anycast IP，在 DNS 查询（UDP）时，利用 BGP 网络的选路策略找到具体的主机。一般 BGP 会采取就近路由选路的方式，因此不同于按照 Local DNS 选择 NS 的策略，因为不是所有的 Local DNS 都采用最短的 RTT 时间进行 NS 选择，所以在 NS 多点部署策略不能达到最优效果时，Anycast 利用路由层面的寻路，自动化地找到较优路径。当某一个 NS 出现问题时，BGP 能够自动找到其他路径进行自动容灾（路由收敛），这也解决了 Local DNS 选择 NS 带来的问题，即当某个 NS 不可用时，某些 DNS 软件仍然会将 UDP 包发送给不可用的服务器，需要人工介入。当然 BGP 的容灾时间非常不可控，它取决于公网路由的收敛时间，如图 6-7 所示。

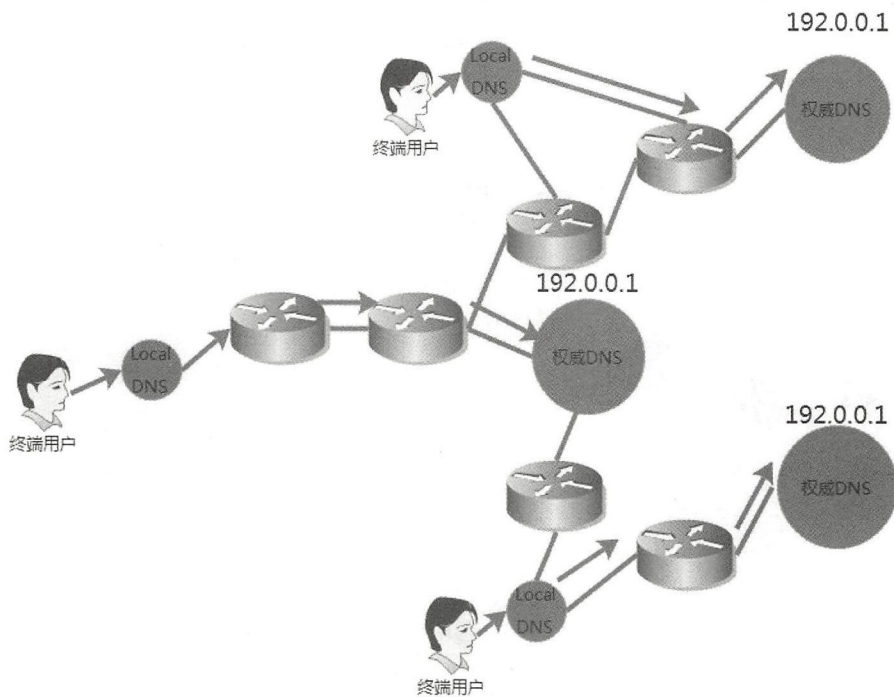


图 6-7

6.3 总结

在分析 DNS 问题并进行优化的过程中，主要的思路如下。



1) 发现问题

需要对 DNS 进行监控，通过监控发现问题。

监控的关键是要真实化、细分化、常态化，从来没有一劳永逸的优化方案，要与时俱进地做常态化的监控和细分。特别是监控需要贴近用户真实的体验，只有监控做到真实才能确定“问题”是否是真正的问题。

2) 分析问题

DNS 的基本原理总的来说是比较简单的，只有对原理不断了解和深入，才能知道如何分析问题的本质，只有看到问题的本质才能提出比较合适的解决方案。在 DNS 的问题分析过程中，要学会使用各种工具分析问题，方便高效地发现问题。能够提出 DNS 解决方案，需要对 DNS 的迭代过程、CNAME 的基本过程、经历的所有环节、每一个迭代用了哪个 DNS、DNS 分布在哪里等，一清二楚。

3) 解决问题

DNS 优化的关键在于，能够提供就近部署解析的架构，让用户可以就近部署解析。设置合理的缓存是优化的后手。

DNS 的优化过程如图 6-8 所示。

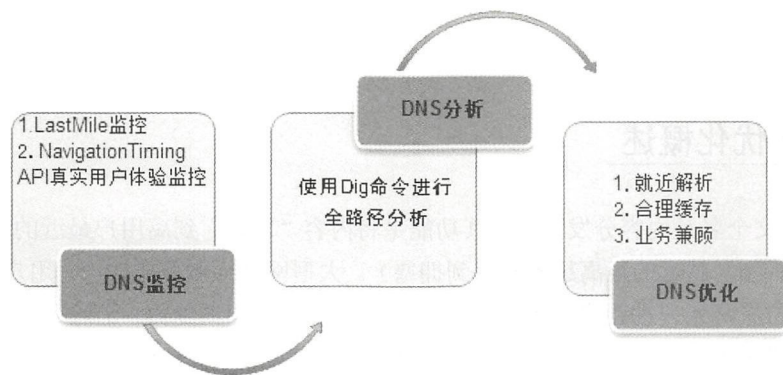


图 6-8



7

第 7 章

CDN 优化

7.1 CDN 优化概述

CDN 的中文全名是内容分发网络，其功能是将内容“发布”到离用户最近的服务器上，有效地避免网络拥塞（越远的距离越容易遇到拥塞）。大型网站一般分布较广，用户地域跨度大，而网站机房的位置离用户的距离有远有近，CDN 提供就近访问的能力，消除了由于用户离机房的距离不一样带来的体验差异，是大型网站不可缺少的基础组件，CDN 的优化对于提供高性能的用户体验起到了关键的作用。

CDN 在大型网站构建过程中，伴随着网站从小到大，出现的问题也会比较多。需要根据 CDN 的一些特性结合网站自身的实际情况，通过一定的手段，才能达到最优的使用效果。

本章从 CDN 的基本原理出发，介绍一些 CDN 常用的优化方案，并通过多个实际比较深入的案例，讲解在 CDN 使用过程中需要特别注意的问题，读者可以将其作为前车之鉴，避免犯



同样的错误。

CDN 对于一个大型网站而言，主要提供了 6 种能力。

1. 静态加速能力

通过本地化缓存加速能力给用户提供一个尽力而为的就近访问的高性能访问架构，将用户访问的内容缓存在边缘节点上，消除由用户地域差异而导致的用户体验不一致，提供不同地区用户的相对一致的高性能访问体验。

2. 卸载源站能力

CDN 将资源缓存在它的服务器上，访问是在用户和 CDN 之间进行的，原来用户的直接请求都发送到网站服务器上，移交到 CDN 上后，源站的访问量和带宽占用都会大幅度减小。特别是对大型网站而言，图片等静态资源占了网站所有请求的 90% 以上。图片访问量对于大型网站来说是巨大的，服务器要提供具备相应吞吐能力的服务，其架构设计、运维规划、监控和预警要十分完善，否则很容易出现稳定性问题。后面将会介绍 CDN 命中率突然变低，造成源站出现各种不稳定的问题；也可以看到，CDN 的命中率对于减小源站的压力十分关键。总而言之，CDN 的存在大大减小了源站的压力，提高了网站的稳定性。

3. 防攻击能力

一般比较成熟的 CDN 提供商至少有数百个 CDN 节点，甚至数千个，而把资源放在 CDN 上，对网站的恶意攻击大部分都会将目标放到 CDN 节点上，CDN 是一个天然的跨地区甚至跨洲的大型分布式系统。大量 CDN 节点的存在，可以有效地将攻击由中心化分散到 CDN 的边缘上，从而有效地阻止或者减小攻击造成的危害。

4. 动态加速能力

CDN 提供静态加速能力的原理是通过将资源缓存在 CDN 边缘节点上，让用户访问资源的网络距离变短，从而实现性能的优化。CDN 不仅适用于可缓存资源的静态加速，而且可以用于动态请求的加速，其原理是通过 7 层路由路径的优选，克服 BGP 选路的缺点，实现动态加速能力，后面将详细介绍 7 层路由优化的基本原理和实战经历。

5. 用户访问序列优化能力

CDN 能够做的事情超乎我们的想象，CDN 如果能获取网页的 HTML，就可以实现访问序列的优化。

1) 页面内资源预取

获取网页的 HTML 后，CDN 服务器会将 HTML 中的图片、CSS 和 JS 标签的资源提前预取



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

到 CDN 的边缘节点上，等浏览器下载序列到相对应的资源时，边缘节点上已经存在该资源，从而避免回源站处理并获取资源，减少性能损耗。

2) 页面间资源预取

页面间资源预取指的是在用户访问当前网页时，将要访问页面的资源提前预取到 CDN 边缘节点上，等浏览器下载序列到对应的资源时，避免回源或者从 CDN 二级缓存节点上获取资源，以减少网络延迟。

6. 定制化模块开发能力

CDN 不仅提供各种标准功能，而且提供定制化的功能开发，这些功能模块中有不少已经标准化，例如边缘化的图片压缩、边缘化图片格式转换、自适应图片下载等功能。

7.2 CDN 的相关术语

1. 边缘服务器 (Edge Sever)

对于边缘服务器，CDN 提供了就近访问的能力，边缘服务器节点就是实际提供给用户就近连接、访问的服务器。

2. CDN 命中率

CDN 一般提供的是静态加速能力，静态加速能力通常通过缓存架构来实现，CDN 命中指的是 CDN 服务器有该资源缓存存在，请求到达 CDN 节点时，CDN 服务器可以在本地缓存获取资源直接返回给客户端，如果没有命中，则需要通过 CDN 节点到源站获取资源。CDN 命中的概率即 CDN 命中率。

3. 回源

当 CDN 没有命中缓存时，需要到源站去获取资源，这个过程称为回源，回源需要从 CDN 节点层层代理访问，最终到源站获取资源。

4. 中间层服务器 (Midgess Server)

边缘节点比较分散，因此存在缓存穿透的问题。为了避免回源引起的性能大幅下降，在 CDN 的中间层服务器中将多个 CDN 节点的访问进行收敛，从而大幅提高命中率。

5. L2 Cache

L2 Cache 也是 CDN 的中间层服务器，通常也称 Midgess Server 为二级缓存。L2 Cache 通常空间更大，是多个 CDN 边缘节点的收敛层。Edge Server 被称为 L1 Cache，L2 Cache 层是 CDN



架构避免回源的常用组成部分之一。

6. 卸载率 (Offload Rate)

CDN 命中率通常也称为卸载率，表示卸载源站访问的程度。

7.3 从应用看 CDN 的基本原理

CDN 实现过程非常复杂，应用 CDN 技术来做设计方案，不需要把 CDN 的所有细节都搞清楚，但是需要知道 CDN 哪些能做、哪些不能做，用了 CDN 后会带来什么问题等。了解 CDN 的原理，能够更好地帮助我们做设计，更好地促进业务的发展。

7.3.1 CDN 基本架构

用户访问图片的一般路径如下：

- (1) 在 DNS Lookup 时，由全局负载均衡器调度将离用户近的节点发给用户的浏览器。
- (2) 浏览器与 CDN 的边缘节点建立 TCP 连接，并将请求发送给边缘服务器。
- (3) 如果边缘服务器没有命中缓存，会将请求代理给 CDN 的 L1 Cache 节点。
- (4) 如果 CDN 的 L1 Cache 服务器也没有命中，会将请求继续发送给源站。

注：CDN L1 Cache 节点通常将相邻的几个 CDN 节点请求进行收敛，当同一资源在一个节点预热过，又从另外一个节点访问时，可以在 L1 Cache 层命中缓存，避免回到源站去获取资源，从而避免性能大幅下降。

(5) 源站接收请求，并通过源服务器的处理，给 CDN L1 Cache 服务器响应，直到最终将响应逐层返回用户端浏览器。

简单地说，用户浏览器获取图片或者静态资源的一般过程是，先通过 CDN 的全局负载均衡器的调度获取离用户“最近的”CDN 节点的 VIP（虚拟 IP），并建立 TCP 连接，发送请求给 CDN 的边缘节点，如果在 CDN 节点上没有命中缓存，请求会逐步传递给最终的源服务器，并最终由源服务器回应给用户浏览器，并在各个 CDN 节点上建立各自的本地缓存。

7.3.2 CDN 全局调度

CDN 全局调度的基本结构和全局负载均衡器的实现原理比较简单，它能够接收在域名查询



请求时，根据用户 IP 地址的（Local DNS）来源，返回给 Local DNS 和用户 IP 地址区域匹配的 CDN 边缘节点的 VIP。通常一个地区有两个 CDN 集群节点，两个 CDN 节点同时给不同的用户提供服务（容灾和负载均衡）。为了防止 DNS Cache 过长，导致一个节点过载，通常 CDN 节点的 TTL 控制在一分钟以内，当一个节点发生问题时，可以在较短的时间内切换到另外一个节点。

常见的 CDN 全局调度器有 F5 的 GTM 设备和其他智能 DNS 软件。全局调度器的基本结构如图 7-1 所示。

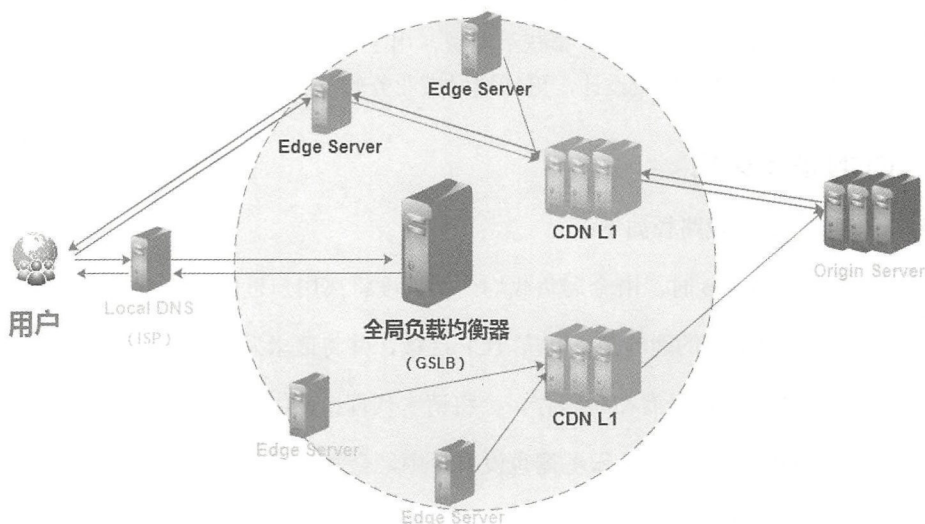


图 7-1

全局调度器从本质上说是一个智能的 DNS 解析工具，无非是做得更加智能、功能更加强大。全局调度器主要有 3 个元素：Region（区域）、POOL（CDN 节点池）、Member（CDN 节点 VIP）。当域名查询请求发送给全局调度器时，全局调度器能够匹配用户的 IP 地址（一般是 Local DNS 的 IP 地址）和 Region 的对应关系，找到对应的 POOL，根据 Score（分数）的大小，找到合适的 POOL，再通过 POOL 找到对应的 Member，Member 里面的 VIP 根据配置的权重，最终向用户返回 CDN 节点 VIP。过程比较容易理解，但是如何做到更加合适、更加稳定，妥善处理各种异常情况，例如负载单点过重的情况、不可用的情况等，做到 100%高可用，并不简单。

7.3.3 CDN 基本调度方式

CDN 提供的是尽力而为就近调度的架构，并不一定是就近调度。在实际商业运转过程中，



需要根据客户的要求和实际情况，给予不同的调度方案。CDN 厂商需要在成本和体验之间做出权衡，例如 CDN 提供商在美国有数千个集群，而在乌克兰、中国台湾才 20 个节点，为了节省成本和容量，CDN 提供商会将流量调度给美国的 CDN 节点，让用户浏览器进行访问，这实际上和 CDN 厂商号称的就近访问有很大的不同，但是商业上就是如此，必须做出权衡，在成本投入上做出选择。

1. 基于 Local DNS 的静态调度

对 DNS 的基本原理有所了解，才能做好 CDN 的解决方案，CDN 提供的产品只是工具，如何结合自己的业务使用 CDN 同样十分关键。

静态调度是 CDN 调度的基本方法，全局调度器根据 Local DNS 的 IP 地址（或者终端机器的 IP 地址，EDNS 协议），在其配置里面找到对应 IP 地址所在的 region，并最终找到合适的 CDN 节点，如图 7-2 所示。

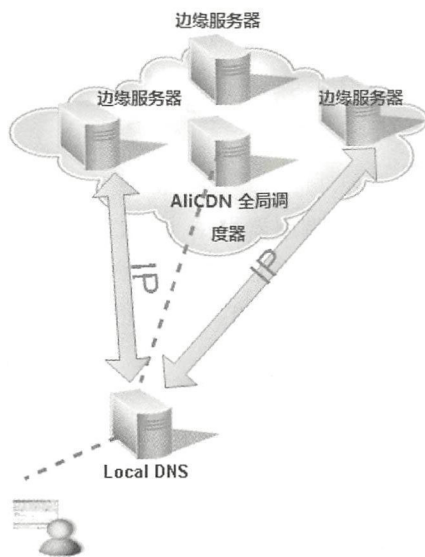


图 7-2

2. 基于 RTT 的调度

基于 RTT (Round Trip Time) 的调度，是全局调度器根据 Local DNS 的 IP 地址，将预调度、多个候选的 CDN 节点和 Local DNS 之间的 RTT 进行比较，全局调度器会将 RTT 短的 CDN 节点调度给用户，如图 7-3 所示。

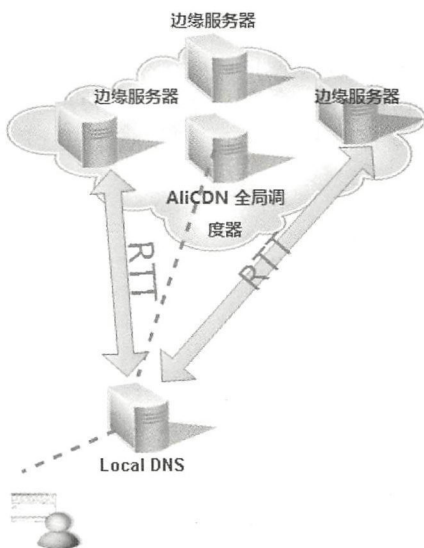


图 7-3

3. 基于成本和带宽的调度

基于成本和带宽的调度是指全局调度器根据 CDN 节点出口带宽的大小来决定使用哪个 CDN 节点进行访问。基于成本的调度是指 CDN 厂商从全局出发，在某些业务少的地区，调度器将访问调度给 CDN 全局分布点较多的地方。因为往往不会考虑在业务少的地区投入过多的硬件成本（自建或者租借机柜、服务器、带宽）。

4. 基于服务等级的调度

基于服务等级的调度是指，CDN 服务提供商为了保障服务等级更好的网站客户的访问，通常会将网络延迟更小（用户访问延迟小）、运行更加稳定的节点给这批用户。例如 eBay 和 Amazon 同时购买了 CDN 的服务，但是 Amazon 购买了服务等级更高的用户许可协议，这样 CDN 会提供响应更好、网络延迟更小的节点给 Amazon，甚至为了绝对保障服务等级高的客户的利益，CDN 提供商会强制将一定百分比的流量调度给机器充足的远距离节点进行访问。

上面是最常见的 4 种调度方法，CDN 的调度通常是上述调度方式的结合，在成本和容量足够的情况下，一般是就近选择 RTT 调度或者静态调度。但是 RTT 调度不容易实现和做到精确调度，特别是在首次访问的时候，没有大量的样本数据的积累，很难判断哪个 CDN 节点离用户最近、RTT 更短，同时为了保障更高级别客户的利益，对于低级别的客户，会强制牺牲一部分用户的用户体验。





综合这么多因素考虑，CDN 调度确实是非常复杂的，CDN 在平衡上的把控，很难做到最优，对 A 合适，对 B 不一定合适，CDN 的使用一定要逐步调优，这就是笔者一直强调的，对于 CDN 绝对不能仅停留在使用层面，还需要特别清楚里面的运行机制，这样才能有策略地部署监控，找到问题的数据依据，最终把自己的业务调整到最佳状态。

7.3.4 CDN 加速的基本实施流程

如果想让某个资源或者某个域进行 CDN 加速，一般要经过下面几个流程，如图 7-4 所示。

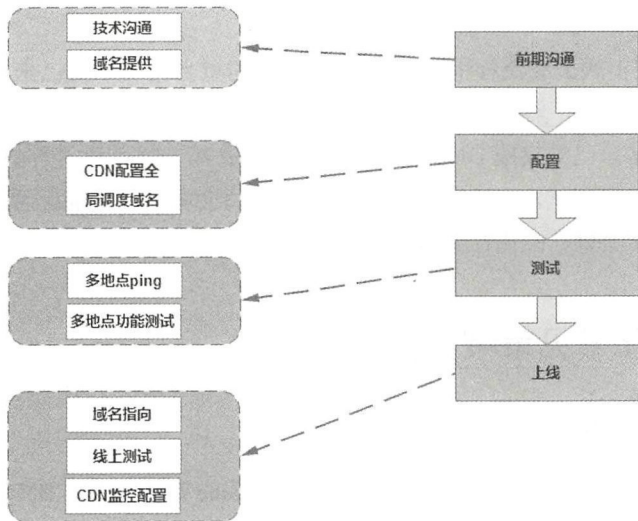


图 7-4

1. 前期沟通

主要包括：license（用户许可协议）的沟通，不同的 license 级别价格不同，不同 license 级别提供的功能也有差别；容量的沟通，CDN 提供商会检查容量和带宽是否足够；网站关注的国家和地区的沟通，如果是一个国家，需要给出重点保障的省市，CDN 提供商会初步验证这些重点保障的地区是否有问题。

初步沟通完成之后，需要提供 CDN 加速的域名，并同时提供源站的域名，以便在 CDN 没有命中时回源使用。

2. 配置

CDN 提供商会根据预加速的域名和回源域名，配置好调度域名，并提供给网站测试人员。





3. 测试

CDN 提供了全局调度域名后，可以 ping 这个域名获取调度后最终的边缘服务器地址，如果需要测试多地不同 CDN 节点的表现，可以使用第三方专业性能监控公司提供的 instant test 里面的 ping 工具，从不同的区域 ping 全局调度器，即可获得多个 CDN 边缘服务器的 IP 地址。获取 IP 地址后，注意最好用 curl 命令进行测试，直接使用 IP 地址在浏览器中访问通常会被拒绝。curl 命令如下：

```
curl -H "Host:www.gtms.com" "HTTP://121.10.0.56/tps/i2/TB1Ia.gif"
```

4. 上线

测试完成后，在正式上线流程中，一般 CDN 加速的域名由网站自己来申请（也可以用 CDN 的域名，需要另外收费），然后由 CNAME 给 CDN 的全局调度器。所以 CDN 的调度过程，首先还是需要经过网站自己的权威 DNS，再由权威 DNS 配置一个别名，指向 CDN 的全局调度器。再到线上进行真实功能测试，测试完成后，CDN 提供商会将监控功能配置好，监控后台可以获知该域名的命中率、流量、点击次数等。

7.4 CDN 优化常见策略

7.4.1 静态化缓存优化

CDN 提供的最基本功能是静态缓存功能，所以 Cache 服务器是 CDN 软件系统的标配。从软件架构上来说，一般分为 3 层，如图 7-5 所示。

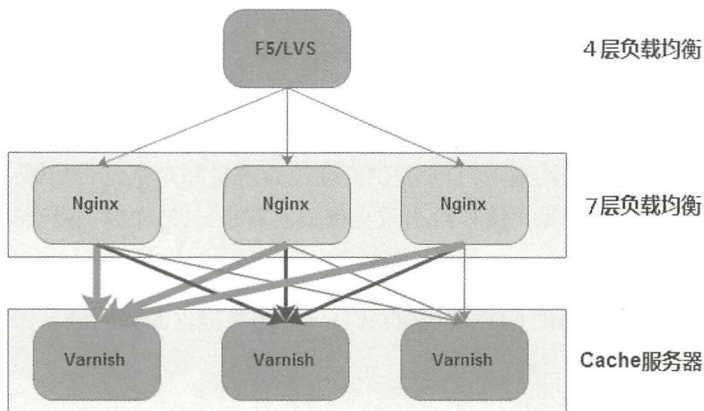


图 7-5





第一层：高效的 4 层负载均衡。

众所周知，LVS 和 F5 提供了高效的 4 层负载功能，4 层的效率高主要在于解包的速度快，在 OSI 7 层网络模型中，TCP 在第 4 层，当 HTTP 请求包发送到服务器时，基于 4 层的负载均衡能够快速将包转发，并且起到均衡负载的作用。

第二层：基于命中率而设计的 7 层负载均衡。

该设计可以针对 HTTP 相关的属性进行负载均衡，如 Cookie、URL、Method，甚至 Parameter，而 CDN 一般针对的是 HTTP 访问网站的加速，所以 7 层负载均衡能够根据 HTTP 报文里面的内容进行负载分担，分担 LVS 通过 NAT 转发过来的数据包。如果直接到缓存服务器，由于 LVS 一般只提供有限的负载均衡方法，例如 Roundrobin，或者通过 5 元组（客户端 IP 地址、port、目标 IP 地址、port、LVS 地址）哈希到真实 IP 地址上，不能保证高命中率，同一个 HTTP URL（如图片），不能保证一台缓存服务器的命中率特别高，特别是对一些近似长尾的访问。为了解决这个问题，中间架设了 7 层转发服务，它能够根据 URL 运用哈希算法，从而保证同一个 URL 基本上都在同一个服务器上访问，进而保证了最终的命中率。

第三层：本地化 Cache 服务器。

能够提供本地化 Cache 能力的服务器非常多，如 Varnish、Squid、Traffic Server 等。由于 Varnish 性能好、稳定性佳，得到了广泛的应用，特别是相对比较老的 Squid，性能要高出几倍，稳定性更佳。

7.4.2 动态内容静态边缘化

通常 CDN 静态加速的对象是图片、JS、CSS 等静态资源，静态资源具有可以被缓存的特性，在一定时间内不会更改，对实时性要求也比较低，同时也比较容易通过修改 URL 来获取页面的最新内容。由于动态页面本身（如 JSP、ASP、PHP）的特别要求（如业务打点，统计 PV、UV）、实时性（如商品价格）等，一般都需要从源站获取内容，但是在某些场景下，页面大部分的 HTML 内容（JSP、ASP、PHP 服务端渲染和执行后的结果）可以缓存（或者局部可以缓存），而实时的部分可以用 CSI（Client Side Include）或者 ESI（Edge Side Include）来实现。动态 HTML 因为涉及首屏、白屏等用户体验相关的内容，所以动态内容如果能放在离用户比较近的地方，可以大幅提升用户体验。

1. CSI

所谓的 CSI 其实就是通过浏览器端发起 Ajax 请求，从源站获取实时性和一致性要求高的数





据，例如价格信息，网页的其他部分全部可以缓存到 CDN 的边缘服务器上。典型的电子商务的场景是商品详情页面，页面的大部分 HTML 内容都可以缓存，价格的信息要求强一致性，因此可以将获取价格的信息重新改造成 Ajax，只要设置 max-age 就可以确定 CDN 缓存时间（如果只想在 CDN 上缓存，不想在浏览器本地缓存 HTML 内容，可以使用 s-maxage 响应头）。

从图 7-6 可以看出，CSI 是通过浏览器客户端获取实时性要求高的数据的。

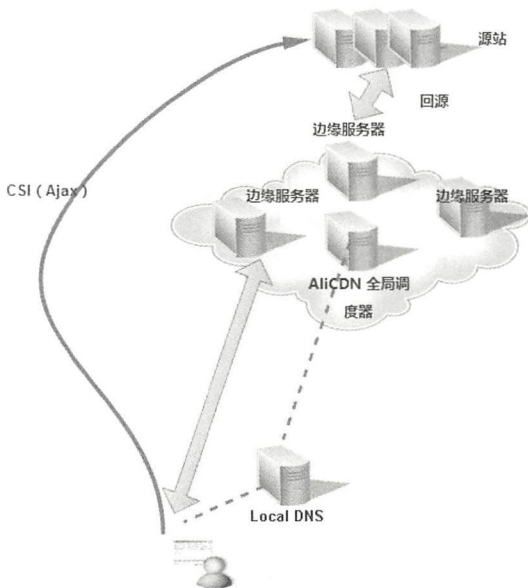


图 7-6

2. ESI

ESI 通过使用简单的标记语言来区分可以缓存和不可以缓存的片段并进行描述，不可以缓存的部分通过边缘服务器实时地从源站获取，可以缓存的部分从边缘服务器本地缓存中获取，并在返回给用户端浏览器之前，组装成完整的 HTML 给浏览器。通过这种控制，可以有效地减少从服务器抓取整个页面的次数，只从源站中提取少量不能缓存的片段，因此可以有效地降低源站的负载，同时缩短用户访问的响应时间。ESI 是一个简单的标签，代码如下。

```
<html>
<head>product detail</head>
<body>
<esi:include src="getProductPrice.htm?productId=11111"/>
</body>
</html>
```





第一次 CDN 缓存没有命中时，从源站获取 HTML 内容（源站不渲染 ESI 标签的内容），获取如上的 HTML 内容后，将内容缓存到本地，再将 ESI 标签 SRC 属性中的 URL 解析出来，并发起对这个 URL 的访问，源站响应的内容和本地的内容进行拼接和组装后返回给浏览器。第二次访问时，CDN 边缘服务器只要将 ESI 标签的内容解析出来，并发起请求将动态内容和本地缓存中的内容组装后返回给浏览器。

ESI 实现的 HTML 内容的缓存由 CDN 的边缘服务器负责动态内容和静态缓存中的内容的组装，如图 7-7 所示，当页面要依赖 SEO 引流时，必须使用 ESI 技术，因为搜索引擎的爬虫无法获取异步 Ajax 的内容。

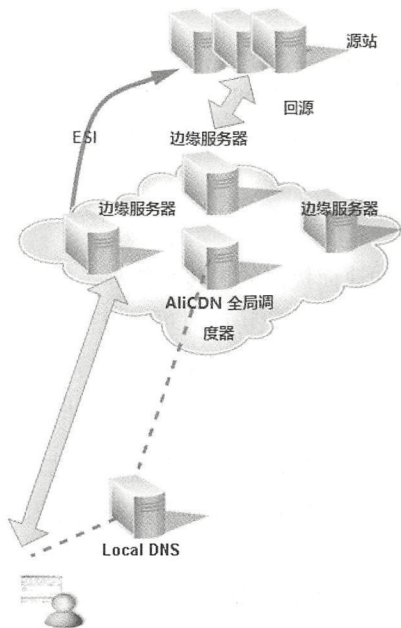


图 7-7

3. ESI 和 CSI 的使用场景

CSI 为了满足实时性要求高的业务要求，通常将实时性的内容通过浏览器端发起请求到源站获取内容，相比 ESI 来说，用户能更快地看到网页内容。ESI 实际上有一部分内容需要回源，缓存的内容在 CDN 的边缘服务器，响应给浏览器的内容是在 CDN 边缘节点或者二级节点进行拼接的。在容错方面，ESI 更强，如果回源部分的请求出现 404 错误或者 500 错误，CDN 上可以处理成相应的错误码和内容给浏览器。例如，在发生 404 错误时，CDN 需要忽略缓存内容，直接响应给浏览器源站响应的 404 错误内容。CSI 是 CDN 先将缓存的内容响应给浏览器，然后





浏览器发起实时的部分请求，再通过修改 DOM 节点内容进行更新。但是一旦 Ajax 请求发生问题，页面的内容不会发生变化，如果是关键内容，如折扣信息或者价格信息，将无法弥补。另外，如果实时性要求很高的内容通过 CSI 来获取，网页会出现闪烁的效果，特别是源站返回慢的时候，这种问题将更加明显。

综合来说，ESI 更适合于对重要性内容需要实时化和强一致性的内容静态化的解决方案，CSI 的实时处理部分通常适合对网页重要内容不做更改的情况，如业务打点请求。使用 ESI 还是 CSI 需要考虑对用户体验的影响、实时部分的重要性，以及容错处理能力。但是 ESI 的处理过程相对复杂，改造成本相对较高，源站需要自己实现 HTML 的拼接逻辑，并做各种容错处理。

综上所述，两者比较如下：

- ESI 实现成本较高，容错能力强，页面加载性能较差。
- CSI 实现成本较低，容错能力弱，页面加载性能较好。

7.4.3 动态加速优化

动态加速的概念是相对静态加速而提出的，传统的静态加速一般指的是，通过 CDN 的边缘服务器（Edge Server）缓存资源，让用户访问时可以直接到离用户最近的服务器上获取缓存的资源。所以在静态加速过程中，资源具有可以被缓存的特性，也就是说，在缓存的时间内，只需要回源站获取资源一次，然后就可以到边缘服务器上获取资源，从而起到加速的作用。

使用动态加速进行优化时，用户的请求仍然会发送到源站，它将通过某种方式减小源站端到用户端的网络延迟。在网站访问过程中，大家都知道从请求发送给服务器、服务器响应，到浏览器接收，大部分时间都消耗在网络上，服务器端的耗时一般比较短，在机房内部的处理，无论是 CPU 的处理时间还是机房内部的网络转发，丢包率都比较低，耗时都比较短。这和报文传输机制相关，无论是发送报文还是接收报文，都需要经过网络层、TCP 传输层，网络转发需要经过 BGP 路由选路的过程，实际的地理距离和网络距离存在很大差距，这会造成网络转发的延迟时间变得比想象中要长很多。TCP 传输带来的网络延迟和 TCP 处理机制相关，特别是 TCP 的拥塞处理机制，以及 TCP 为了保持可靠性而设置的 TCP 重传机制。TCP 一般都是在 TCP 重传定时器溢出时重新发送丢失的报文，而 TCP 重传定时器的超时时间如果很长，同样会造成很大的网络延迟。

从本质上说，无论是静态加速还是动态加速，大部分都通过减小网络延迟来进行加速。静态加速是通过把资源缓存在离用户最近的地方来实现优化的目的，动态加速是通过优化传输、优化网络转发等方式来实现优化的目的。



静态加速一般都是静态文件，如图片、JS、CSS、HTML，这些类型的资源是允许有缓存的，在缓存时间内，内容很少或者几乎不发生变化。而一些动态请求，如强一致性的读/写操作等，都是不能缓存的。这个加速属于动态加速范畴，所以动态加速的请求具有每次请求回源的特性，每次请求都会到达源站，不会改变用户到机房的地理距离，但是会减少网络上的距离。

实现动态请求的加速有很多方法，如前面所述，目标都是通过某些方式减小网络延迟，常用方法如图 7-8 所示。

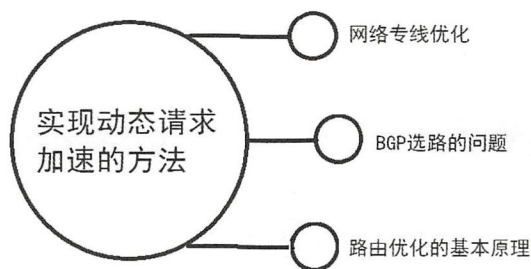


图 7-8

1. 网络专线优化

在用户离机房距离非常远时，特别是全球化跨境电子商务网站，距离长会造成网络延迟大，例如美国和乌克兰的传输时间达到 200ms，比淘宝国内的用户访问淘宝的服务器的延迟大很多。专线的优势是具有更小的网络延迟和更低的丢包率，这两个特性能够减小 TCP 因为重传而导致的超大的网络延迟，网络延迟也更加固定。

2. BGP 选路的问题

实现动态加速的非常重要的方式是路由优化，这是网络路由的特性决定的，网络路由并不是按照最近的原则选路，这给路由线路的优化留下了空间。这个空间不是故意预留的，而是因为全世界网络的互联涉及众多的运营商，网络的路径由众多的网络运营商主导完成，各个运营商之间的网络流量的相互流动、运营商之间的流量流动的不对等，会造成一些小的运营商不得不在考虑成本的情况下进行路由选择。例如，中国网通从中国电信的网络走的流量，比中国电信从中国网通走的流量大很多，中国电信可能会向中国网通收费，中国网通可能为了减少费用而限制流量或者绕路路由。限制流量会造成丢包，绕路会造成网络延迟变大，这就是流量不对等引起的运营商成本变高，这个看起来人为的因素会造成 BGP 选路的延迟变得很大。

综上所述，流量互通的不对等是 BGP 路由选路的网络延迟大的主要原因，特别是跨境的网络路由转发更容易出现这种问题。通过测试发现，巴西离加拿大的距离大概为 8000km，按照光



速的传输速度，传输一个 TCP 包需要 27ms 左右，实际测试发现网络延迟达到 200ms。这就是 BGP 选路引起的网络延迟。

3. 路由优化的基本原理

大家都知道路由层面的东西完全是由运营商控制的，路由优化不是对众多的运营商网络进行的，而是利用 CDN 边缘节点组成的虚拟网络节点来进行优化，所以一般来说，只有 CDN 这种规模的节点部署能力才能达到路由优化的目的。路由优化是利用 CDN 节点 7 层代理转发改变 BGP 选路的路径来进行优先选择的。其最佳路由线路的探测过程如图 7-9 所示。

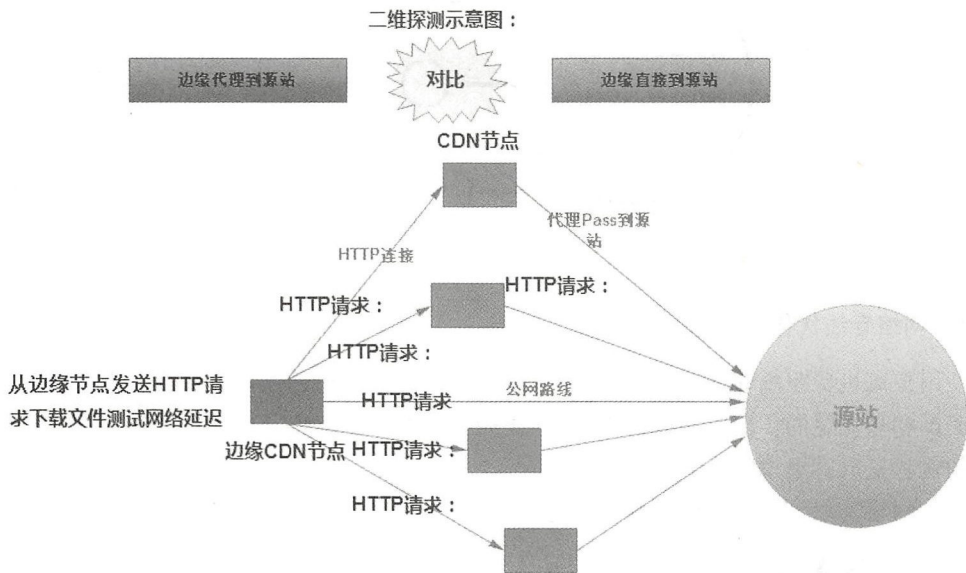


图 7-9

(1) 源站会按照 CDN 提供商的要求提供一个大小合适的资源，资源大小在 20KB 以上（测试丢包率）。例如图片，一个静态页面最好是和网站本身的访问资源的大小相匹配。

(2) CDN 会在源站周边选择一批候选 CDN 节点。

(3) CDN 在边缘节点尝试从源站下载指定的资源，下载请求首先发送到指定的候选 CDN 节点，然后在 CDN 候选节点修改成源站的域名，通过这种多路探索选择一条最佳的路由线路来进行路由优化。

(4) 最佳线路需要考量多个指标，如丢包率、RTT、整体下载时间，这些指标用来判断线路是否是最佳的。在实际的 CDN 厂商实现的过程中，远比上面简要描述的原理复杂得多，上



面介绍的只是两层路由优化，实际实现过程远不止两层，而且路由优化中的探索是动态进行的。在数万台服务器之间，探测出一条比较合理的路径并不简单，没有很好的模型支持，很难做到兼顾大部分的加速需求。还是那句话说得对：纸上得来终觉浅，绝知此事要躬行。知道这些知识的原理，优化思路会得到扩展，在网络优化遇到瓶颈的时候，可以想到用这个方案来解决问题。目前国外主流的 CDN 厂商都有这种技术，如 Akamai、LimeLight、CDNNetwork 等，在国内阿里的 CDN 动态加速技术已经得到广泛的应用。

7.4.4 用户序列优化原理

用户序列优化是 CDN 根据用户请求的 HTML 内容和浏览器下载的顺序进行提前解析，将浏览器需要的资源提前预取到 CDN 节点上，等到浏览器下载该资源时，该资源已经提前在 CDN 节点上了，如图 7-10 所示。

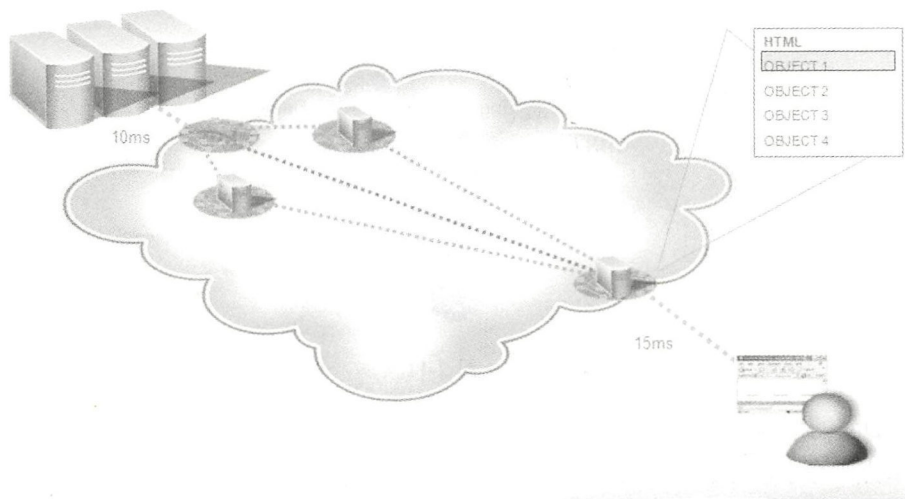


图 7-10

7.4.5 域名合并优化

域名合并是常见的 CDN 优化手段之一，对于 Web 网页来说，为了提高浏览器的并行下载能力，往往会使用多个域名。CDN 在缓存配置上有很多策略，能够根据业务的变化进行灵活配置，而域名合并是其中一种优化方式。

例如下面的 URL，在未调整之前，Cache key 是整个 URL，缓存不能重用。域名合并之后，

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

Cache key 是除了域名之外的 URL，两个图片缓存可以重用，从而命中率得到提升。

- HTTP://z00.aaaa.com/support.jpg
- HTTP://z01.aaaa.com/support.jpg

7.4.6 多级缓存架构优化

用户发起请求到边缘节点的 VIP1 集群（L1），如果在 VIP1 集群上没有，会到同一地区、城市的 VIP2 集群上获取，如果 VIP2 集群也没有，就回源到 L2……直到 L3（称为 Parent）。总之海外 CDN 的回源路径是“Edge（L1）→Peer（L1）→L2→L3”，如图 7-11 所示。

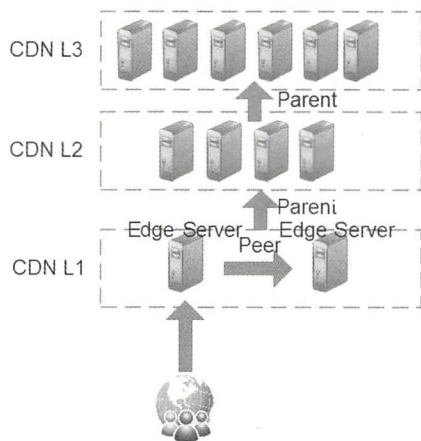


图 7-11

从 CDN 的架构来看，比较优秀的 CDN 提供商做了很多优化，例如 Akamai，如果请求未在边缘节点上命中，那么就到同一个区域与 Edge 节点对等的 Peer 节点上获取。这是比较优化的做法，因为大多数 CDN 节点，Edge 节点和 Peer 节点的距离很近，回源的网络延迟会很小。因此在 CDN 架构上，加 Peer 层对减小网络延迟有比较大的帮助，特别是在命中率不够高的时候。

7.4.7 301、302 跳转边缘化访问和多终端边缘化判断

网站都有大量的 302 跳转的逻辑存在，例如很多网站对于 PC 访问、Mobile 设备访问使用统一的域名。当 Mobile 设备访问时，采用 301、302 跳转到 Mobile 站点，跳转从浏览器端发起，因此至少会浪费一次 RTT，对性能会有较大的影响。用 CDN 代理域名，可以在 CDN 的边缘服务器上跳转，从而让用户网络访问链路变短，提升性能。

7.5 CDN 优化实战

7.5.1 CDN 的不合理架构造成 304 请求耗时长优化实战

1. 问题现象

有一段时间，接到不少用户反馈说，打开页面缓慢，有时候 1 分钟页面都没有加载完毕。出问题的资源平时访问量并不大，而且热点不如买家端的页面，接到投诉后需要先把问题重现。果然，这个问题可以重现，并且问题现象如下。

页面的某些 JS 下载时间在 5s 以上，如图 7-12 和图 7-13 所示。

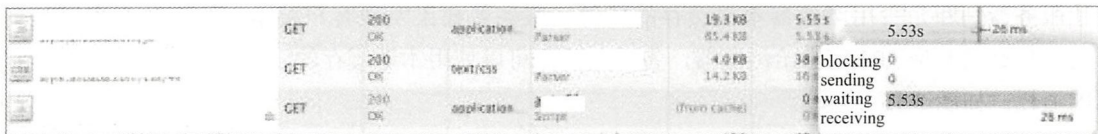


图 7-12

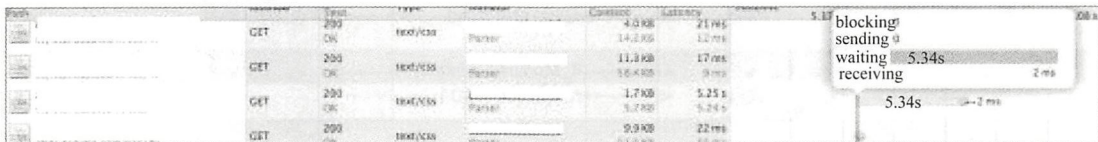


图 7-13

2. 问题的初步分析

一个静态资源耗时长达数秒以上，先要对静态资源的访问路径进行分析，找到链路上的瓶颈，另外可以根据相似性原则进行问题排查。所谓相似性就是问题同时有多个现象，但是这些现象后面的原因有可能是一样的，所以从一个现象入手同样可以发现问题的瓶颈。从本现象中可以看到，304 请求耗时比较长，那么就从 304 请求问题入手。

第 1 步，逻辑分析，确定问题的基本方向。

从图 7-14 的用户访问链路中可以看到：

- (1) 边缘服务器网络延迟小，用户离边缘服务器近。
- (2) CDN Edge Server 到 CDN L2 Cache 的物理距离远小于 L2 Cache 离源站的距离。
- (3) L2 Cache 到源站物理距离远，而且需要从分布式文件系统中读取文件。

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

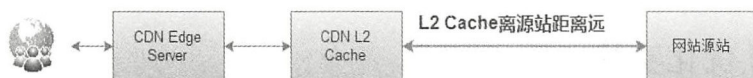


图 7-14

由逻辑分析得出，回源非常可能是造成 304 请求和资源下载时间长的主因。

第 2 步，原理分析，进一步确定瓶颈。

分析 304 请求问题为什么会有如此长的耗时，如果是回源引起的，304 请求问题会引起回源吗？

先回顾一下 304 请求问题的基本原理：浏览器在访问同一资源时，会向服务器发送请求，让服务器判断是否用浏览器本地缓存的资源，服务器如果发现客户端本地缓存的资源是最新的，那么会响应 304 请求给浏览器，告诉浏览器可以使用本地缓存的资源，这样可以减小网络的消耗，如图 7-15 所示。

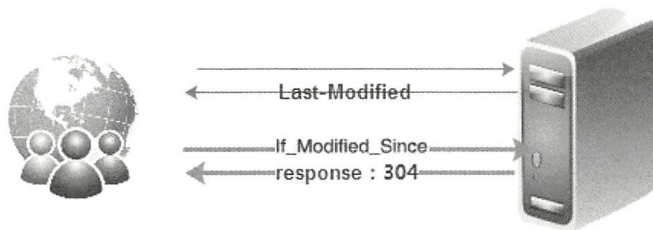


图 7-15

详细过程如下。

浏览器第一次访问某个资源时，会在响应头里面加入 Last-Modified 字段，标识此资源在服务器上的最后更新时间。当浏览器第二次访问同样的资源时，在请求头里面加入 If-Modified-Since 字段，这个字段的值是第一次请求服务器给浏览器的 Last-Modified，服务器以这个时间和文件的最后更新时间做比较，如果浏览器记录的时间比服务器记录文件的更新时间早，说明文件内容已经发生了修改，服务器会给浏览器最近的资源，如果文件内容没有更新，则给浏览器 304 的响应，不带文件内容。所以从 304 类型的请求来看，由于没有文件内容需要下载，因此耗时长长的原因可能跟 CDN 的特殊处理过程有关系。

既然 304 类型请求是轻请求，那么需要深入查看 304 类型请求在 CDN 中的处理过程是怎样的，如图 7-16 所示。

第一次请求某资源的时候，CDN 全局调度器会调度一个离用户近的 CDN 边缘节点，然后

经过 CDN 的 L2 Cache 节点、网站源站获取该资源，并缓存在各自的本地，包括浏览器本地。

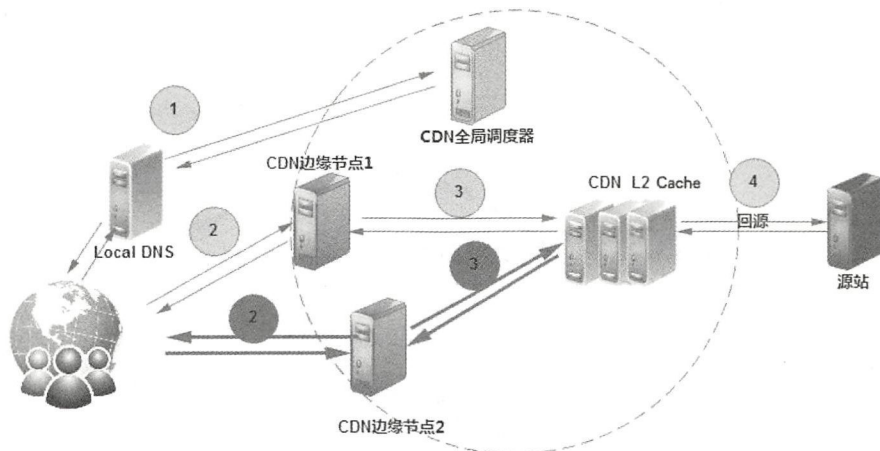


图 7-16

在第二次重复请求该资源的时候，CDN 为了负载均衡的设计和需要，通过设置足够短的 TTL（一般在 30s 以内），将负载均衡分配给另外一个 CDN 边缘节点，所以只要第二次访问和第一次访问调度给不同的 CDN 节点就会出现下面两种情况。

- 第一种，如果在第二个 CDN 节点上没有请求的资源，那么请求会回源到 CDN 的 L2 Cache 去获取，如果 L2 Cache 上没有资源，那么请求继续去源站获取，直到边缘节点上存在此资源文件，再把获取图片的 Last-Modified 时间和请求头带过来的时间进行对比，如果发现客户端的文件是最新的，服务器给客户端 304 请求类型的响应。
- 第二种，如果在第二次调度的 CDN 节点上有请求的资源，但是浏览器本地的资源比 CDN 节点上的资源还新，此时需要从 CDN L2 Cache 节点获取资源，如果 CDN L2 Cache 上也没有，那么回源站获取。

这里有一个问题，第一次访问时，CDN L2 Cache 上已经有该资源的 Cache 了，有没有可能在第二次访问的时候发生 CDN L2 Cache 资源丢失，然后回源站获取？

第 3 步，对 CDN 架构深入分析，定位 CDN L2 Cache 在什么情况下会丢失缓存。

从图 7-17 可以看到，CDN 的内部结构如前面所描述，第一层是 4 层高效的负载均衡，用于快速转发请求，第二层是 7 层负载均衡，用于 URL 哈希均衡负载，并将相同的 URL 转发给后面的第三层如 Varnish 之类的本地缓存服务器。CDN L1 层就是同一地区的两个 CDN 节点，CDN L2 层是多个 CDN 节点组成的庞大的集群，提供请求收敛层，提高命中率。一个用户从

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

CDN L1 层的 LVS1 第一次发起请求 1，命中 CDN L2 层的 LVS1。另一个用户从 CDN L1 层的 LVS2 发起同样的资源请求 2，命中了 CDN L2 层的 LVS2。此时资源尽管被重复访问，但是却在 CDN L2 层丢失了缓存资源。

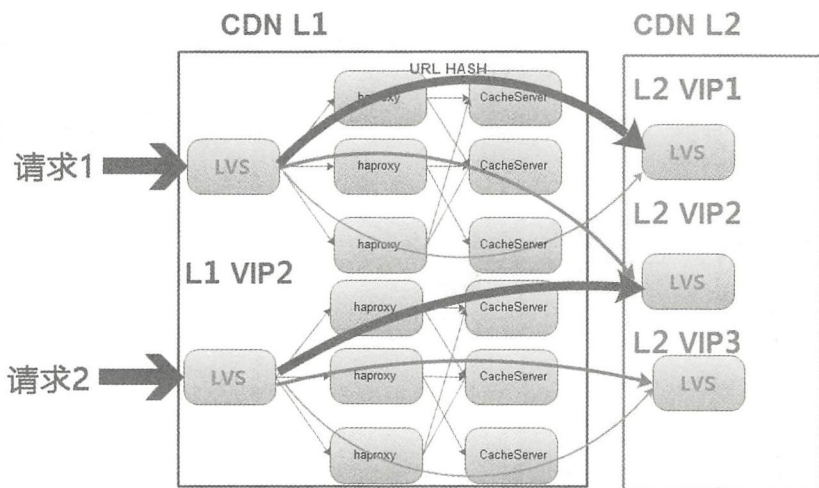


图 7-17

在出问题前，CDN L2 层部署了 20 个 VIP 集群，也就是在后台这种访问类型的页面访问频率不高，Max-age 设置时间不长就会出现这种情况，20 个 VIP 集群，命中率只有 5%。

注：CDN L2 层如此设计有一定的合理性，CDN L2 层在避免回源的架构上和源站卸载上起到了非常关键的作用，L2 层节点越多，可用性越好。对于 L1 层来说，配置 L1 层的回源拥有同一个域名的 L2 层，所以 L2 层必须是同一个域名，实现起来同样比较简单，由 L2 层的全局调度器分配给 L1 层一个就近访问的 L2 层节点。

第 4 步，解决问题，L2 层节点收敛，提高命中率。

从上面可以看出，在 CDN L2 层节点部署过多会造成 L2 层对访问过的资源仍然执行 Miss Cache。对于本例中的情况，在访问频率不高、热点不够的情况下，对 CDN L2 层的 VIP 节点进行收敛能够大幅提高命中率，如图 7-18 所示，将 CDN L2 层收敛成 2 个节点，这样命中率至少提高到 50%，加上静态资源还是有一定的热点的，所以命中率调整之后，L2 层可以剩下大部分从 L1 层被执行 Miss Cache 的资源。

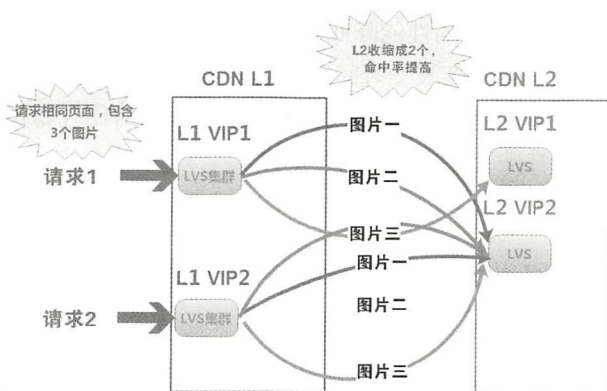


图 7-18

7.5.2 静态资源命中率优化实战

伴随着网站从小到大的发展,CDN 曾导致网站多次发生问题。网站开始上 CDN 时只有 30% 的命中率, 经过两年的优化, 最终将 CDN 命中率提高到了 95%。经历过才知道, 要合理地使用 CDN 才能让业务收益最大化。

1. 命中率优化实战之一 ——整体命中率从 40%提高到 70%

几年前, 笔者所在网站发生了一件比较大的事情, 各个业务指标全面下降, 如图 7-19 所示。

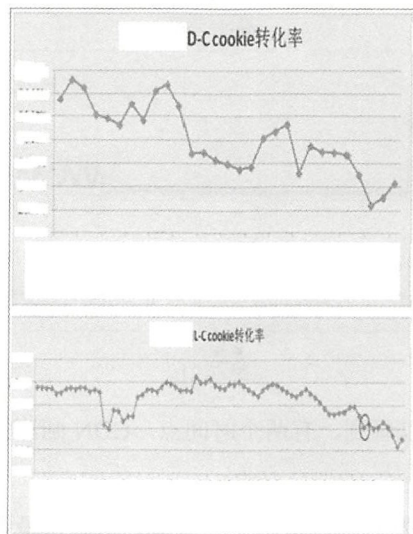


图 7-19

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

工作人员很快跟踪到重要页面的性能降低问题，分析全站性能有如此大的下降，初步判定和站内的通用性能组件相关，同时把衰变时间点确认出来。谈到通用组件，首先就想到了 CDN，CDN 的性能降低通常和大量回源相关，因为 CDN 节点本身相对比较稳定，特别是网站使用国外顶级的 CDN 提供商，在性能和可用性方面有保障。

1) 源站回源分析

将 CDN 源站访问情况的监控拿出来，看看从源站上能不能发现流量大幅上升，如图 7-20 所示。

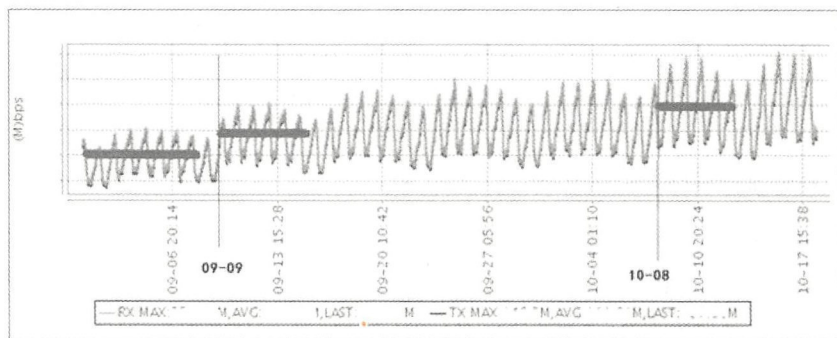


图 7-20

从源站上看，由于大量的回源引起了 CPU 的消耗大量增加，如图 7-21 所示。

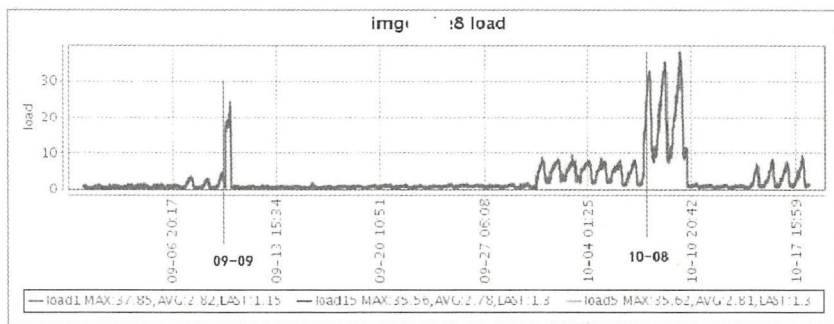


图 7-21

从图 7-20 和图 7-21 中可以看到，有两个时间点，CDN 回源的流量明显增加，初步确定这两天网站可能发布了什么项目。

于是把衰变时间点对应的发布项目拿出来一一进行排查，最终注意到一个项目：系统工程师为了紧急处理上线需求的问题（由于没有购买 CDN 提供的 SSL CDN 服务，导致发布上线的



SSL 应用大量出错)，解除了 CDN 对 style.aaaaaa.com 域名的托管。最终导致 CSS 和 JS 相关的请求直接去源站获取 CSS 和 JS，这些重要的请求全部没有经过 CDN 的静态加速，最终用户的请求通过远距离传输到达源站，从而导致性能大幅下降。

恢复 CDN 加速后问题来了，CDN 的命中率有所增加，从图 7-22 可以看到，CDN 在正常时命中率为 35%~40%，恢复之后为 27%~30%，问题并没有彻底解决。

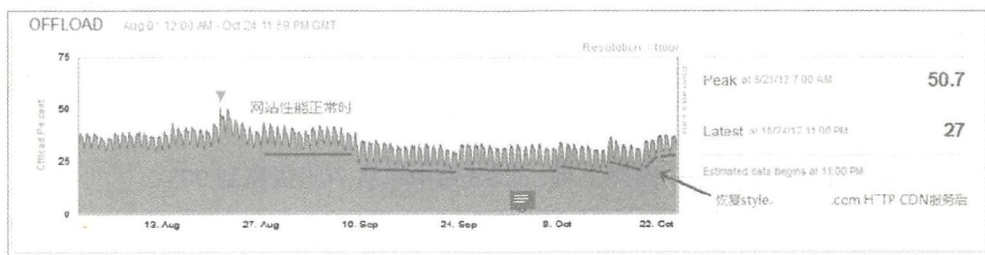


图 7-22

2) 问题再定位

如果想要知道 CDN 没有命中的原因，那么必须在源头上下工夫，CDN 加速的是图片、CSS 和 JS，CSS 和 JS 的问题在上面的步骤中得到了解决，所以问题可能出在图片上，如图 7-23 所示（图中敏感内容做了处理）。对于电商类型的网站而言，图片大都来自于商品，图片的大量回源可能和下面几种情况相关。

- 图片新增尺寸。
- 有大量新图片产生。

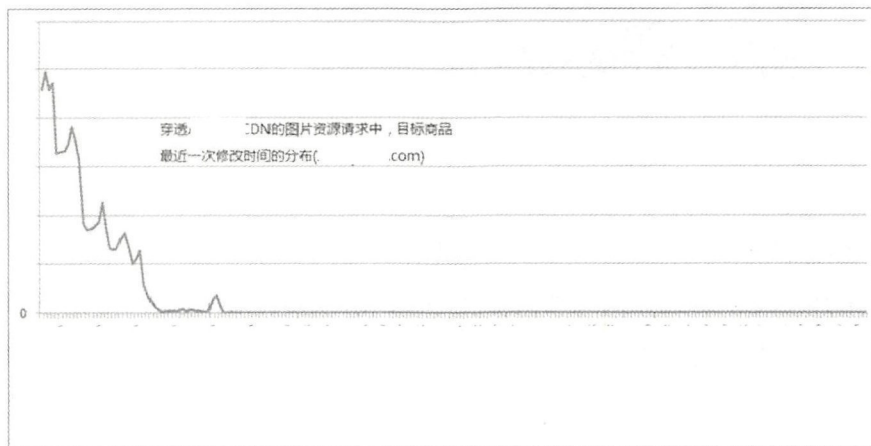


图 7-23



3) 回源问题的解决方案——多级缓存架构升级

在 2012 年的时候，AliExpress 使用的 Akamai CDN 只有一级缓存，缓存命中率只有 30% 左右，随后由于访问量和大量长尾冷访问的商品数量增加，导致 L1 层的命中率大大减小，使得源站的压力大大增加、性能大幅下降，业务数据表现变差。后来 Akamai CDN 使用了多级缓存架构，2012 年调整的多级缓存架构使得命中率从 40% 提高到 70%。而当时的架构是典型的二级缓存架构，对于全球化的网站而言，把多个地区的访问收敛于一个大的缓存层，对于缓存的命中率提升相当关键，它可以把一级缓存未命中的请求穿透到 L2 层，L2 层重用了 L1 层穿透的请求，显然命中率会提高很多。

2. 命中率优化实战之二——图片、JS、CSS 命中率从 70% 提高到 90%

上述优化让网站的命中率从 40% 提高到了 70%，但是 70% 的命中率仍然没有达到理想的状态，只要命中率稍微变化一点，源站的访问仍然会大幅增加，稳定性也受到一定的考验。

二级缓存的命中率对于低热度近似长尾的数据访问，仍然有穿透到源站的可能。所以建立了 L3 层，将近似长尾的访问收敛于 L3 层，这样可以大幅卸载源站的压力。为了进一步提升性能，网站使用 Akamai 的 license 升级，针对命中率提升，Akamai 加了三级缓存 L3 层，可以把 L2 层漏掉的访问收敛于 L3 层，在 L3 层可以命中，这样图片的命中率从 70% 提高到了 90%，如图 7-24 所示。

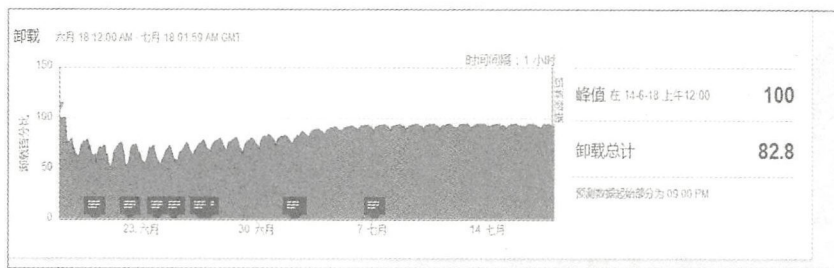


图 7-24

3. 命中率优化实战之三——JS、CSS 命中率从 65% 提高至 100%

三级缓存的上线给图片的命中率带来了很大的变化，但是对 JS、CSS 等静态资源并没有很大的影响，后续一直想在命中率上继续优化，却找不到很好的方法，也尝试过从源站的 URL 日志进行分析，由于那时大数据工具还不够完善，因此分析源站的日志也没有起到很大的作用，要想发现问题，一定要先有监控数据，有足够细分的数据支撑，才能确定问题到底在哪里。

CDN 内部架构和工作机制对缓存的命中率有极大的影响。我们在 CDN 使用初期采用的是



粗放式的使用方式，所有域名都在一个 CPCode 下，CDN 提供商以 CPCode 为最小单位统计命中率。在这种情况下，不知道哪个域名的命中率低，当时网站有 20 个以上的静态域名，到底哪个域名、哪类 URL 命中率很难获知。监控细粒度化非常必要。于是把几个重要的域名拆分成不同的 CPCode 进行监控，很快就发现了问题，如图 7-25 所示。

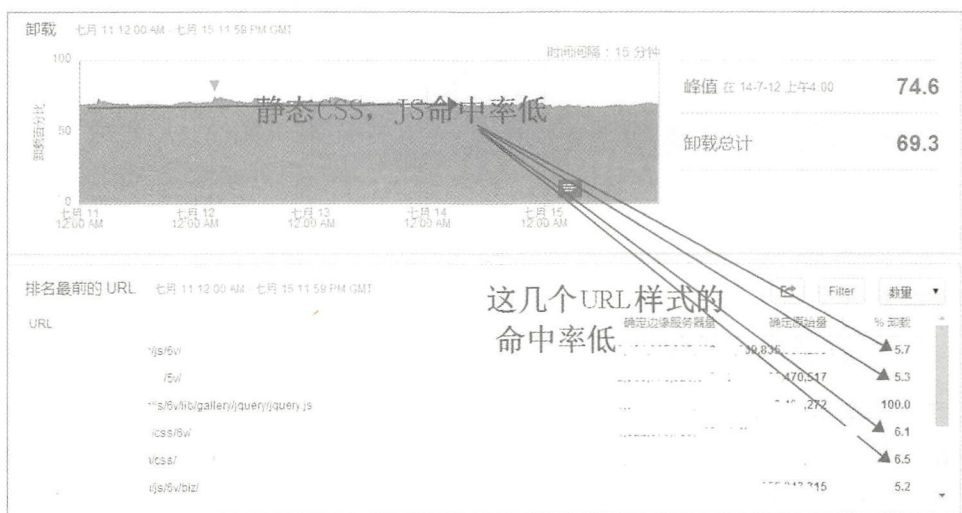


图 7-25

发现类似/js/6v 这样的 URL 命中率极低，于是和 CDN 的售后一起追查这个问题，把日志拿出来分析，最终发现是网站配置的 URL 规则和 CDN 默认的规则不一致导致了这种类型的请求被 CDN 当作动态请求而回到了源站去访问，从日志上可以发现回源请求中很多都有“?” 这种标记的 JS 文件缓存没有 hit:

```
HTTP://style.bbbbbb.com/css/v/??ap/core/core.css,run//home.css,run/font.css,run/w/buy/comp/newuser-popup.css,run/buy/comp/countryFlag_s.css,run/buy/module/footer.css,run/buy/module/header-all.css?t=33326e4e8_f54df2051
```

为什么带有这种标记的请求会被回源？原来 CDN 缓存的时候要区分动态请求和静态请求，对于动态请求 CDN 默认会回源站请求，通常这种区分在 CDN 内部有默认的配置，根据请求 URL 的扩展名来决定是否将请求回源，详细过程如下：找到请求 URL 中的“?”，“?” 前面就是文件的扩展名，如果不带扩展名，CDN 系统默认这个 URL 属于动态请求，那么直接穿透到源站。如上面的 URL，当发现有两个“?” 前面没有任何的扩展名时，CDN 默认它属于动态请求，不缓存，直接将请求回源。

用 CDN URL 匹配规则，将“??” 任意匹配“?” 改成一个“?” 进行精确匹配，这个



URL 就属于 CSS 文件（如上例），就可以缓存了。

这个规则修改之后，命中率大幅提升，如图 7-26 所示。

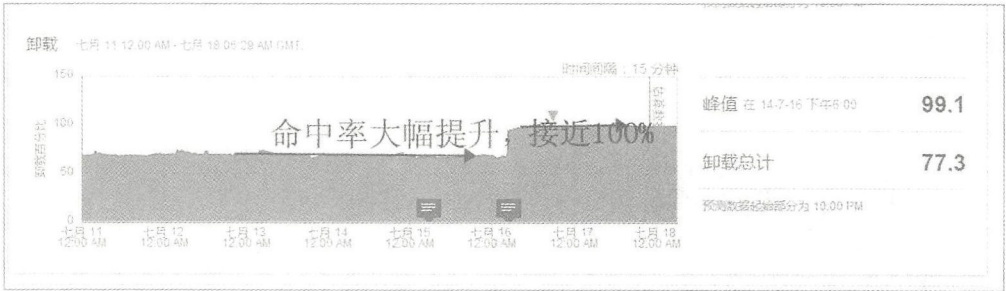


图 7-26

7.5.3 CDN 动态加速优化实战

1. 问题描述和分析

几年前碰到一个案例，由于用户访问一个页面需要数十秒，因而使用户体验受到了极大的影响，从用户端录屏来看，发现以下问题：

- 动态请求耗时长达 2s。
- 302 类型跳转多。
- 服务端处理时间短（不超过 200ms）。

从瓶颈分析来看，网络耗时占整体耗时的大部分，如图 7-27 所示。

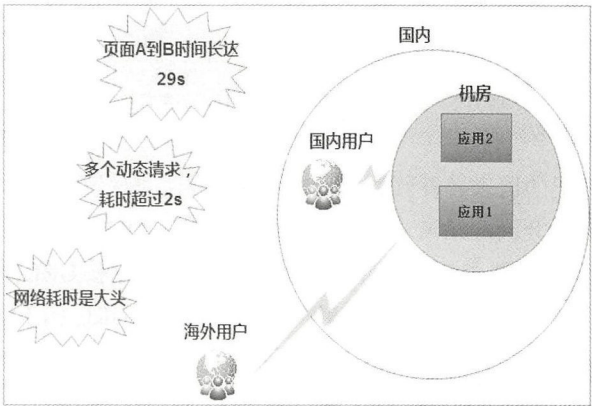


图 7-27



2. 解决方案

对于跨境的访问，由于地理距离长、网络结构复杂，非常有可能造成网络延迟大，另外对 TCP 传输来说，有如下两个非常重要的现象。

1) 越大的文件越容易丢包

在实战过程中，可以明显看到这个规律，笔者发现 Ajax 请求多的应用 TCP 重传率很低，HTML 级别的访问，TCP 重传率要高很多。

2) 越长的距离越容易丢包

距离越长的访问，网络转发时经过的路由器也越多，网络拥塞的概率也越大。

对于跨境的访问，最大的问题还是路由的问题，运营商之间的对等路由交换、流量上的不均等，会导致默认的 BGP 选路并不是最佳路径。BGP 选路从来不会承诺选择网络延迟最小的线路，BGP 选路最基本的承诺是路由可达，路由线路的优劣直接影响网络延迟的大小，而动态加速方案能够解决路由选路的问题。

3. 动态加速方案带来的问题

1) 调度问题

动态加速方案是为了加速中国台湾用户到淘宝服务器的访问速度，但是最直接的问题是，其他用户也会访问到同样的域名，如果相同的域名都加速，会导致其他用户也会从第三方 CDN 的网络走，而第三方 CDN 并没有足够的 CDN 节点，从而无法制定好的路由方案。解决这个问题有两个方案。

方案一：域名分割，修改逻辑，独立域名交给第三方 CDN（Akamai）进行托管解析。

(1) 修改域名涉及测试回归，成本巨大，目前有 3 个域名，而且修改 buy.***.com 的域名走 Akamai 基本上是不可行的，原因是涉及大量系统的改造。

(2) 独立判断逻辑，涉及底层的改造，开发成本巨大。

方案二：根据区分用户 IP 地址决定是否给第三方 CDN 进行加速。

DNS IP 库判断是否托管，根据用户 IP 地址来判断走不走 Akamai CDN 托管，如果来自于非中国台湾用户，就走原来的逻辑，如果来自于中国台湾用户，就走 Akamai CDN 托管解析，访问链路由 Akamai 进行控制，以便进行 TCP 优化和路由优化，该方案优点如下：

(1) 程序不用修改，测试基本无须回归。



(2) 域名不用修改，几乎不需要开发成本。

该方案基本原理如图 7-28 所示。

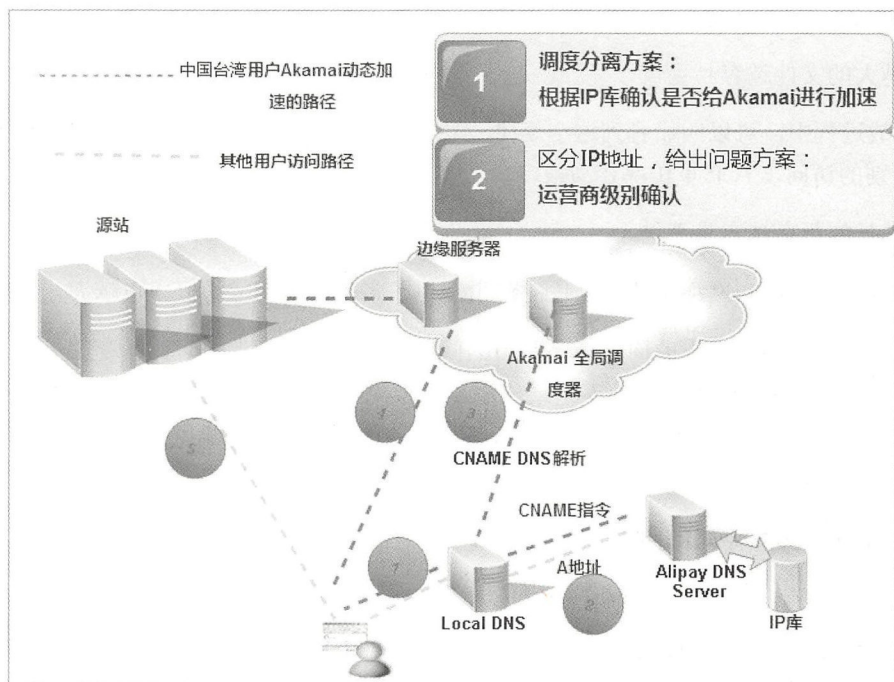


图 7-28

方案二的问题是，用 IP 库来区分用户访问很难做到精准。

方案二看起来比较好，只要 IP 库精准，就能在 DNS Lookup 的时候判断是否 CNAME 给第三方 CDN 的全局调度器，这样对应用方来说是透明的，无须做任何修改，但是如果区分用户 IP 地址有误差，仍然会有一部分非中国台湾用户访问 Akamai 的网络，这部分用户就会受到影响，毕竟淘宝用户的体量非常大，用户的基数也是非常大的。

为什么根据 IP 库判断不能精准地区分用户？这是因为大部分 DNS 输入的 IP 地址是 Local DNS 的 IP 地址，而不是用户真实的 IP 地址。在这种情况下，用户的 IP 地址所在的地区并不和 Local DNS IP 地址所在的地区相同。

- 当 Local DNS 所在的地区 and 用户实际所在的地区有差别时，调度会出现问题

这种情况经常出现在跨国公司，当用户在一个分公司访问网站，但是 Local DNS 却在公司



的总部时，CDN 的全局调度器就会出现错误的调度。

- CDN 节点本身的调度问题

这种情况最常见的是用户使用公共 DNS，如 Google 的 8.8.8.8，它采用了 EDNS 协议，原本可以把用户的真实 IP 地址给 CDN 的全局调度器，但是如果 CDN 的全局调度器不支持 EDNS 协议，那么它拿到的仍然是 8.8.8.8 的 IP 地址，这个时候调度就会出现错误，当然一般的 CDN 都支持这种协议。

2) 如何避免因为调度问题导致用户的体验变差

因为 IP 地址的原因可能会导致用户被调度给 Akamai 进行加速，而实际上这种“加速”比本身的访问要慢很多。为了减少调度存在的问题，采用的方案是细分调度法，其原理如下。

在域名解析时，AliPay 的权威 DNS 根据用户 Local DNS 的 IP 地址来源决定是否 CNAME 给 Akamai 进行加速，和其他的 IP 地址区分不同的是，第一步进行线上验证时，AliPay 的 DNS 区分是中国台湾的 IP 地址还是非中国台湾的 IP 地址，这个区分更加精细，从而使调度上出现错误的概率非常小。

3) 用户真实 IP 地址的获取

在动态加速过程中，基本原理是通过 CDN 的节点代理寻找一条比较合适的路由路线，当到达源站时原有的 IP 地址获取方式（通过 IP 报文的源地址）必须改变，否则源站获取的是 CDN 的 IP 地址。对于 HTTP 类型的加速，一般 CDN 是在 HTTP header 里面加入 IP 地址的标识，目前业界是存在标准的，很多代理服务器已经支持 x-forwarded-for 字段标识，也有用 ORIG_CLIENT_IP 进行标识的，HTTP 报文在经过 CDN 代理之后，CDN 的代理服务器会进行解包，从 IP 报文到 4 层的 TCP 报文，再到 HTTP 报文，CDN 的代理服务器再将从用户浏览器传输过来的 HTTP 报文的 header 加入 x-forwarded-for 或者其他部分，最后将 IP 地址作为值，以 KV 的形式放到 HTTP header 里面，如图 7-29 所示。

与源站建立 TCP 连接和发送请求的过程是封包的过程，将 IP 地址放在 HTTP header 里面，再封装成 TCP 报文，然后将 IP 报文发送给源站，如图 7-30 所示。

但是对于 HTTPS 的情况就完全不一样了。首先，CDN 代理服务器无法解开户浏览器发送过来的 HTTP 报文，全部是加密过的，CDN 代理服务器无法将 IP 地址放到报文里面。所有的 HTTPS 加速都会遇到这个问题。通常的解决方案是将源站的 HTTPS 证书交给 CDN 进行托管，但是对于支付宝这种安全性要求极高的网站，不会将服务器端证书交给 CDN 代理服务器，这给 HTTPS 的加速带来了难度，如图 7-31 所示。



CDN接收客户端的请求过程

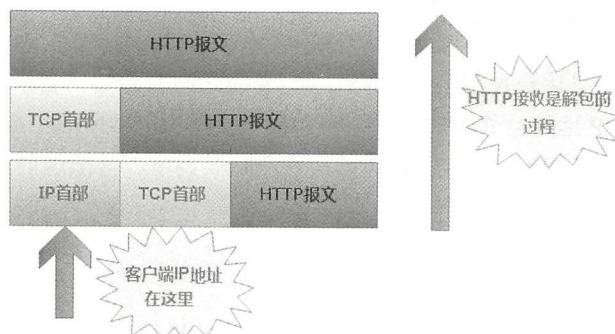


图 7-29

CDN7层转发HTTP报文给源站

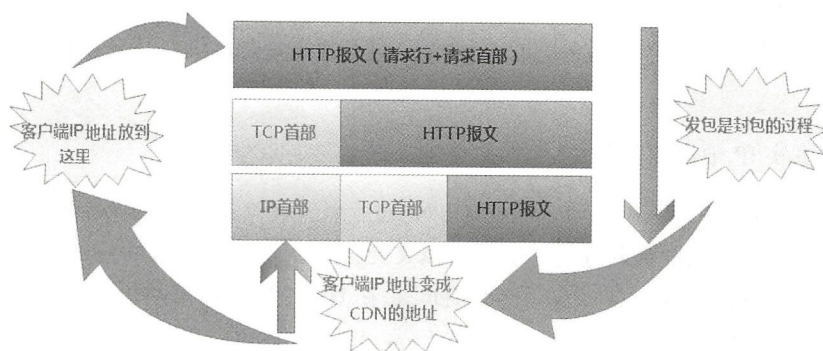


图 7-30

CDN无法解开HTTPS的报文，没有服务器证书
导致客户端IP地址放不进去

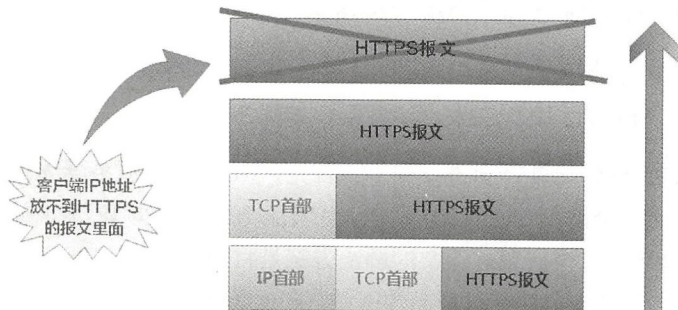


图 7-31

这个问题怎么解决？

如果了解 OSI 7 层协议模型，就能够有解决方案。大家知道 IP 报文才会有 IP 地址，TCP 报文包含端口号，HTTPS 位于 TCP 层之上、HTTP 层之下，而 TCP 报文有 TCP option 字段，如果 CDN 能够将 IP 地址放到访问 TCP 的选项里，那么 CDN 代理服务器就不需要将 HTTPS 报文解开再进行封包传输给源站，问题自然解决了。在发包过程中，将客户端的真实 IP 地址放在 TCP 选项里，报文到达源站的 lvs，经过 fullnat 4 层转发给后面的 RS（RealServer），RS 的 Linux 内核 TCP Kenel 解析 TCP 报文的 IP 地址，在应用层就可以获取客户端的真实 IP 地址，如图 7-32 所示。

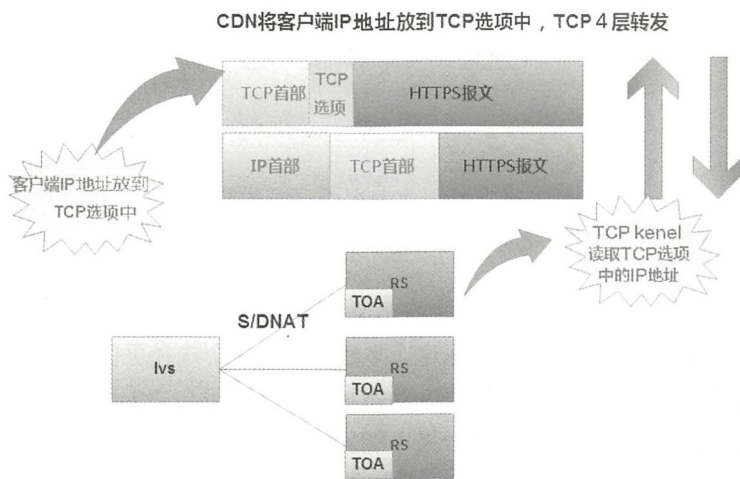


图 7-32

4. 动态加速的问题——白名单 IP 收敛影响加速效果

大型网站在安全性方面要求比较高，经过 CDN 代理之后，源站获取 IP 地址的方式，由从 Socket 中获取转到从 HTTP header 中获取（对于 HTTPS 的动态加速网站，从 TCP option 字段中获取）。大型网站一般都有 4 层、7 层防攻击的能力，为了防止 CDN 被源站防攻击系统拦截，通常的做法是，将 CDN 节点的服务器 IP 列表加入防攻击系统的白名单过滤列表，以防止正常浏览被“误杀”。

但是 CDN 厂商提供的 CDN 节点中的 VIP 列表通常是非常有限的，不是将所有 CDN 节点的 VIP 全部给使用方，这通常是可以理解的，CDN 提供商一旦公布所有的节点信息，其网络拓扑结构往往可以描绘出来，涉及 CDN 的核心竞争力——CDN 的运营商选择及选址。提供有限的 VIP 节点，会影响 CDN 的动态路由效果，如图 7-33 所示。

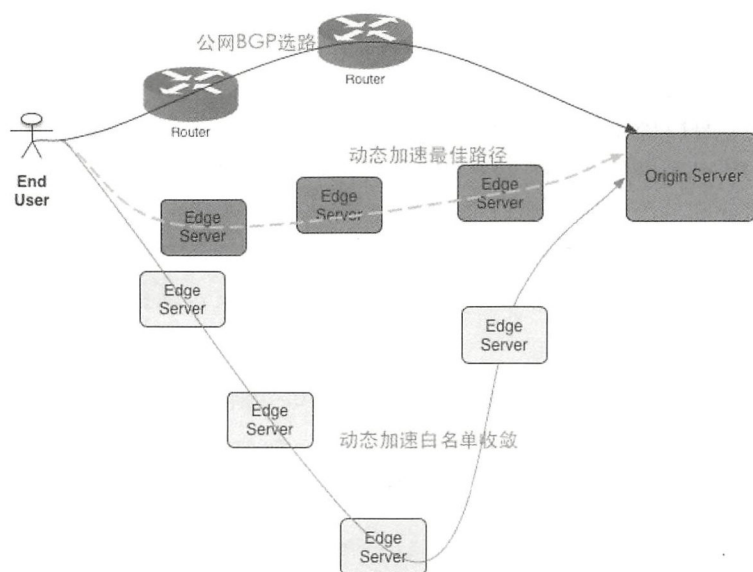


图 7-33

动态加速是利用 CDN 节点的动态路由探测来实现的。其优势是利用了 CDN 的众多节点建成灵活、可变的路由线路，让其可以在这么多线路中有机会选择更优线路。因为 CDN 提供的白名单数量有限，可选择的线路将很少，动态加速的效果将会大大受到影响。从实践来看，加入白名单与没有加入白名单相比加速效果可能差一半，如使用动态加速之后，没有加入白名单动态加速的效果大概提升 10%~20%，但是加入白名单之后，效果提升只有 5%~10%。白名单的方式不仅增加成本（CDN 提供白名单需要额外收费），而且会影响加速效果。

5. 白名单影响加速效果的问题解决

为了获得更好的加速效果，就要解决安全的问题，问题的本质在于源站能否识别 CDN 代理过的请求，如果是 CDN 请求，那么就放过，否则拦截。让源站识别 CDN 代理过的请求，有多种方案。

1) CDN 在 HTTP header 中加入明文标识，源站解析

CDN 和使用方约定标识，CDN 设置标识，源站解析是否有这个标识。对于明文标识一般 CDN 厂商都能够自定义，不需要单独开发，常见 HTTP 代理通常会加入 x-forwarded-for 字段。在 HTTP 报文 header 中加入明文标识，这种方式的优点是对 CPU 资源消耗较小、方案简单，CDN 不用刻意开发。缺点是很容易被恶意篡改、模拟，从而被攻击者绕过 CDN 进行攻击。

2) CDN 在 HTTP header 中加入暗文标识，源站解密

在 header 中通过对称加密算法对标识进行加密。这种方案的缺点是 CPU 资源消耗大，CDN 开发费用也会增加，源站的资源消耗也比较大。优点是安全性高，被恶意模拟的概率小，只要加密 Key 不被破解，很难被恶意模拟。

3) CDN 在 TCP 选项中加入标识，源站解析 TCP 选项

在 TCP 选项中加入明文或暗文，由源站在操作系统 TCP 内核开发进行解析，安全性更高，开发难度更大，也更难被人破解。

这3种方案都在报文中加入标识，缺点是源站都需要开发模块，CDN 也需要修改，小型网站可能没办法使用。

7.5.4 CDN 静态化的问题和优化实战

1. 商品详情页面静态化实战

1) 背景和业务

CDN 主要通过将内容缓存到离用户较近的边缘节点进行加速，所以对于图片、JS 和 CSS 这类纯静态资源，以及 HTML 都可以进行静态加速。通常说的动态页面，很多时候是因为业务需求不同，页面不能被缓存，所以不能通过 CDN 静态加速来解决用户访问性能的问题。但是仍然有很多动态页面是可以被缓存的，只不过业务上能够容忍的缓存时间有限。在跨境业务性能优化的过程中，为了让用户体验更佳，需要解决对用户体验影响很大的两个关键性能指标：白屏时间（StartRender）和首屏时间。根据网页的加载序列，动态请求位于加载序列的第一位，所以动态请求的性能对用户体验的影响很大。

商品详情页面承载商品详细信息业务，包含商品基本信息（商品名称、价格、SKU 信息）、商品销量信息、商品详细信息、评价信息。CDN 静态化尽量选择热点十分突出的页面，比如秒杀商品宝贝页面，商品少、热点突出，CDN 静态化主要带来用户体验的提升，需要有明显的收益。热点极其突出的标准是尽量在 CDN 的边缘节点上命中，一旦到达 CDN 的二级缓存，网络延迟会大大增加，此时静态化带来的收益相比方案的引入带来的复杂性要小很多。

2) ROI 分析

CDN 静态化确实是一个比较好的方法，但是并不是所有的场合都适合。在实施之前，首先要明确具体的优化目标，CDN 静态化可以达到两个目标：其一是提升用户体验，提升首屏加载性能和减少白屏时间；其二是卸载源站的压力，提高吞吐量。

ROI 分析（投入产出分析）是进行策略分析的常用方法，可以避免盲目地进行性能优化。性能优化不是将能够想到的优化都实施到项目中，自始至终地进行 ROI 分析是性能优化策略制定的基本方法。CDN 静态化涉及多个关键问题，这些问题的解决通常会增加架构的复杂性。

对我们的业务来说，一般动态请求通过 BGP 公网选路到达源站并响应，再到 TCP 分段传输，整个过程大概需要 2s。如果优化目标是卸载源站压力，而用户体验提升只是额外的收益，那么 ROI 分析主要看命中率，同时也要看能够卸载源站多少压力。例如，能够减少数百台源站的机器，同时又能缩短白屏时间，那么这个方案就是合适的。以卸载源站压力为主要目标，主要看命中率和减少的机器数量，如果命中率的提升导致减少了几百台服务器，那么这个 ROI 可能是比较高的。

下面通过计算白屏时间的缩短来计算 RTT。一级节点的 RTT 通过实际测量是 800ms，Peer 节点的 RTT 是 1000ms，二级 CDN 节点的 RTT 是 1500ms，源站的 RTT 是 2000ms。

产出分析：根据命中率和 RTT 分析出大概减少的时间，代码如下。

```
<table class="table table-bordered table-strIPed table-condensed">
<tr>
  <td>一级缓存命中率</td>
  <td>Peer 节点命中率</td>
  <td>CDN 二级缓存命中率</td>
  <td>CDN Miss</td>
  <td>减少时间</td>
</tr>
<tr>
  <td>90%</td>
  <td>90%</td>
  <td>100%</td>
  <td>0</td>
  <td> $0.9 \times 0.8s + 0.1 \times 0.9 \times 1s + 0.01 \times 1 \times 1.5s = 0.81s$ </td>
</tr>
<tr>
  <td>30%</td>
  <td>20%</td>
  <td>90%</td>
  <td>5.6%</td>
  <td> $0.3 \times 0.8s + 0.2 \times 0.7 \times 1s + 0.9 \times 0.56 \times 1.5s + 0.056 \times 2s = 1.248s$ </td>
</tr>
</table>
```



```

        <td>20%</td>
<td>20%</td>
        <td>70%</td>
<td>19.2%</td>
<td>0.2*0.8s+0.2*0.8*1s+0.7*0.64*1.5s+0.192*2s=1.376s</td>
</tr>
<tr>
        <td>10%</td>
<td>10%</td>
        <td>50%</td>
<td>45.5%</td>
<td>0.1*0.8s+0.1*0.9*1s+0.5*0.71*1.5s+0.445*2s=1.59s</td>
</tr>
</table>

```

通过上面的分析可以看出，当 CDN 的整体命中率低于 60% 时，性能提升已经不明显，节省低于 300ms 的时间，看看这个数值是不是设定的目标，如果是，那么这个优化意义已经不大，因为 CDN 静态化会带来非常大的架构的复杂性，下面会分析对架构的影响和关键问题的解决方法。因为是秒杀页面，CDN 的边缘命中率基本可以保证在 90% 以上，节省的时间达到 1.2s，这个提升幅度达到 120%，所以秒杀商品宝贝页面的 CDN 静态化从 ROI 上来看是合适的。

3) 关键问题

动态页面的 CDN 静态化是一个完整的架构设计过程，有多个关键问题需要解决。

- 数据一致性和实时性

从业务特点可以看出，商品基本信息实时性要求高，一般需要强一致性（信息一旦修改，展示给用户的信息必须是修改后的信息）。评价信息本身就允许有延迟，从数小时到一天的延迟业务上都可以接受。商品详情信息相对商品基本信息而言实时性要求略低。所以数据一致性问题动态页面的 CDN 静态化的首要问题。

- 同域名非加速页面处理

CDN 是按照域名进行加速的，一个域名一般对应一个应用，一个应用存在多个页面，非常可能存在只加速一个页面的情况，不加速的页面需要让它回源站进行特殊处理。

- 故障快速恢复策略

经过 CDN 静态化加速之后，架构要有容灾的考虑，当 CDN 出现问题时，能够快速恢复到源站访问的能力。

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

- 平滑上线策略

上线需要考虑平滑上线，能够逐步切换上线，平滑上线是重要业务上线的基本要求。从 CDN 的原理上看，CDN 加速都跟着域名走，所以流量切分方案不能选择 Apache Bench test，不能采用源站灰度发布方案。平滑上线是需要考虑的问题。

- 边界业务兼容

经过 CDN 缓存后，会给业务带来另外的问题，主要是 PV 统计的问题。网站实现 PV 打点是根据页面生成一段 JS，这段 JS 发送请求给采集服务器，JS 请求的参数包含用户的 IP 地址、采集时间点等与用户相关的信息。如果这些信息缓存在 CDN 的边缘节点上，就会对业务造成影响。

- 安全考量

经过 CDN 加速之后，安全是必须要考虑的问题，TCP 报文放到源站，变成了 CDN 的 IP 地址，存在被误拦截的问题。

4) 关键问题解决方案

- 域名托管方案

CDN 静态化是通过 CDN 代理来进行的，所以 DNS 解析方式需要从源站的 A 地址改成通过 CDN 来托管解析。最简单的方法是通过源域名的权威 DNS 由原来返回 A 地址改成 CDN 的 CNAME 别名，也就是 CDN 的全局调度器域名，由它来进行调度，并将 CDN 边缘节点的 IP 地址返回给用户端的浏览器。

- 数据一致性的解决方案

秒杀商品宝贝页面中商品的基本价格和折扣需要准实时，库存要保持强一致性。强一致性方案采用 CSI 进行库存判断，如果从源站获取的库存大于 0，那么可以继续下单购买。如果从源站获取库存失败，默认购买按钮是灰色的，只有明确地获取到库存信息才能继续流程，通过这种方式有效地避免了超卖的问题。对于价格和折扣信息同样可以用 CSI 来解决，因为秒杀商品是爆品，命中率非常高，几乎可以做到 100% 边缘命中，因此在折扣达标前后，只要缓存时间足够短（例如一分钟），缓存时间一到，CSI 从源站获取到相应信息并判断是否是最新信息，如果不是，缓存立刻被更新成最新的信息，已经不需要更新 DOM 节点，因此不会出现 Repaint 的现象，页面的闪烁问题几乎不会出现，所以 CSI 几乎不会发生问题。如果使用 ESI 来处理，要麻烦很多，源站改造成 ESI 成本较高，第一次没有命中时，需要实现 ESI 功能，并在源站服务端拼接完整的 HTML，要做 ESI 的容错处理，ESI 架构相对复杂。

- 同域名非加速页面处理的解决方案

在非加速页面的 response header 里面加入 Cache-Control: no-cache，以便 CDN 不会启用缓存机制，保证非加速页面能够被 CDN 识别成动态请求，从而能够到达源站获取响应。

- 故障快速恢复和平滑上线的解决方案

在这里故障快速恢复指的是当缓存内容存在错误或者 CDN 发生问题时，能够随时切换回源站。平滑上线指的是流量能够逐步切换给 CDN 进行加速。故障快速恢复方案可以在故障发生时，将在权威 DNS 上修改 CNAME，改成返回给源站 VIP 作为 A 地址。而平滑上线，根据 CDN 的工作原理和域名解析原理，可以在权威 DNS 上区分国家（可以细分到城市、运营商），按照国家 CNAME 给 CDN 全局调度器。这种流量切分方案需要权威 DNS 根据 Local DNS 的 IP 地址配置地址库，通过判断预设的国家给 CDN 加速。

- 安全保障

由于经过 CDN 代理之后，到达源站的 IP 地址变成了 CDN 的代理 IP 地址，用户真实的 IP 地址被封装在 HTTP header 里面，而原有的源站安全防护默认是根据 IP 报文的地址进行防护的，因此防护策略配置需要进行相应的修改，改成从 HTTP Header 约定的字段进行 IP 地址的获取，同时安全防护需要根据这个 IP 地址进行。但是仍然存在和之前动态加速的 IP 地址一样的问题，即攻击者可以模拟 CDN 的行为，绕过 CDN 进行攻击，因此同样需要加 CDN 白名单，或者通过对称加密的方式和在源站安全防护层进行校验的方式进行安全保障。

2. 用户无法看到对应国家分会场的产品

1) 问题描述

用户反馈对应国家的活动会场的图片无法看到，运营和产品人员将问题反馈给开发工程师：该国用户看到图片非常慢，是不是 Akamai CDN 出了问题？

2) 问题分析

在分析这个问题之前，先看看活动会场商品展示示意图，如图 7-34 所示。

现象：当时运营人员反映的是图片加载慢，拿到用户的截图看到如图 7-35 所示的加载情况，注意转圈的位置，如果是图片加载不了，应该会有文案出现，所以初步判定应该不是图片无法加载的问题。

3) 问题结论

从以上分析可以看出，运营和产品人员的反馈是不对的，实际上是该产品 list 加载的 Ajax 访问不了，因为如果图片加载有问题，区块的文字部分应该是能够展示的。既然是动态内容加

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

载有问题，马上做了以下的动作。



图 7-34

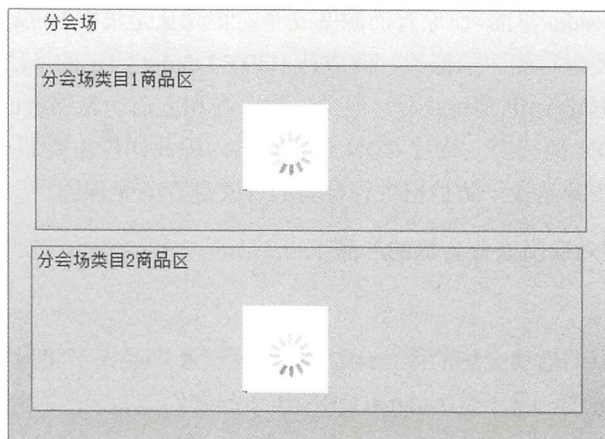


图 7-35

（1）召集开发和产品人员进行讨论，获知是某个域名通过 CDN 进行了动态加速，当时怀疑是 CDN 在该国的加速节点存在问题。

（2）与此同时，把 Ajax 请求拿出来，用 LastMile 提供的即时测试工具看看该国当地用户的情况，发现这个 Ajax 请求需要 15s 才能返回，并且如果直接访问源站（不经过 CDN 加速），只需要几百毫秒，说明 CDN 节点存在严重的问题，如图 7-36 所示。



图 7-36

(3) 为了快速解决问题而不因为分析问题浪费时间，并没有先排查为什么该国的 CDN 节点存在问题，而是启动了应急方案，将 `gaga.aliexpress.com` 的权威 DNS 的路径由 CNAME 给 AliCDN 的全局调度器修改成直接返回给源站的 A 地址，问题马上得到解决。再去测试，发现只要 400ms 就可以返回了。

还有一个问题：该国用户打开图片的速度很慢。

开始看到这个问题时是比较诧异的，因为图片都在 Akamai 上进行了加速，为什么图片打开速度会很慢？

首先猜测是不是 Akamai 出了问题。给 Akamai 现场支持热线打电话，让他们检查是不是 Akamai 出了问题。与此同时，我们从客户机器上 ping `f03.aliCDN.com` 这个出问题的域名，发现 IP 地址有点像 AliCDN 的 IP 地址。这个问题让笔者有点困惑，之前一直以为是 Akamai 进行图片加速的（这个错误的判断为这次问题埋下了伏笔）。这个问题纠结了 10 几分钟，之后用 LastMile 的即时测试工具和 Dig 命令进行测试，发现该国用户用 AliCDN 进行了加速。产品人员登录该国节点的机器发现，这个机器的丢包率是 100%。很清楚了，问题的原因是 `f03` 这个域名使用了 AliCDN 进行加速，而且该国节点是大促一个多月之前上线的，笔者并不知晓。

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

7.5.5 CDN 调度优化实战

CDN 能够提供一致性性能访问的前提是用户访问必须就近调度，在笔者针对 CDN 优化的实战过程中，通过精准监控发现过非就近调度问题。在跨境业务中，由于地区间的物理距离很大，例如中美之间的 RTT 达 500~1000ms（公网），专线的 RTT 是 200ms，俄罗斯和美国之间的 RTT 是 200ms，荷兰到美国的 RTT 是 80ms，所以一旦发生调度问题，CDN 不仅没有加速访问，反而拖慢了用户访问的速度。下面看一个跨境 CDN 调度优化案例。

1. 问题现象和分析

在 CDN 优化过程中，分别发现了 CDN 提供商的两个问题。

第一个问题：A 国家用户被 CDN 调度到 B 国家的 CDN 节点进行访问，对于跨境网站来说，就近访问非常关键，最后 CDN 售后跟踪的结果是，B 国家的节点到 A 国家用户所在的 Local DNS 的 RTT 相对是最优的（相比 B 国家 Local DNS 到 B 国家某个地区的节点的 RTT），当时 CDN 提供商的技术非常先进，调度方式不仅限于静态调度，而且会根据 RTT 动态调度，给用户体验最好的节点，这个问题跟踪了一个月，只能得到这样的结果，但是很快遇到了第二个问题。

第二个问题：在全球化用户体验跟踪的过程中，发现 C 国家的性能比 D 国家慢 30%~50%。C 国家的网络基础环境相对较差，于是部署了 LastMile 监控进行跟踪，结果发现 TCP 创建连接的时间（一般被认为是一次 RTT）比正常的 RTT 长很多。一般来说 C 国家的用户到 C 国家的 CDN 节点的 RTT 不会超过 50ms，但是从 LastMile 监控来看，RTT 很多都在 200ms 以上，从这个结论上首先想到的是，调度可能存在问题。如前所述，CDN 的调度存在多种方式，并不仅是就近调度。因为 Local DNS 的 IP 地址来源于 D 国家，所以导致调度到 D 国家的 CDN 节点把 Local DNS 的 IP 地址提交给 CDN 提供商的技术人员进行排查，结果发现确实有 30% 的 C 国家的用户访问被调度到 D 国家。

CDN 提供商将我们的 CDN 的 license（用户许可协议）进行服务等级升级，修改为更高等级的服务，页面整体性能提升了 10% 以上。

2. CDN 调度问题总结

CDN 调度问题的解决，离不开下面两个条件。

1) 提供足够的数据进行 CDN 排查

第一次排查实际上是失败的，原因有很多，A 国家和 B 国家是相邻国家，地理距离较近，在 RTT 方面和本地调用差距不大。CDN 主要根据 Local DNS 的 IP 地址进行就近调度，如果 Local DNS 的 IP 地址和用户所在的地区不匹配，会导致调度出现错误。当时排查没有成功的主



要原因是通过 WebPagetest 进行测试，不能拿出 Local DNS 的 IP 地址和用户的真实 IP 地址，没有足够的证据证明 CDN 调度出现了问题，也没法证明 CDN 的排查出现了问题。

2) 确保在平时构建良好的常态化的沟通渠道

在 CDN 优化过程中，要和 CDN 工程师之间建立常态化的沟通机制，培养彼此的默契度。忽略沟通，往往在实际操作过程中，会使很多事情的难度增大。实际上在 CDN 的使用过程中，从初始使用到成熟，需要做很多调优工作。CDN 一般会给大客户提供比较优质的服务，大客户与非大客户是根据使用量来决定的，一般大客户也是随着业务的增加从小到大成长起来的，因此在小的阶段需要建立沟通机制，以便在长大的过程中避免在发生问题时找不到合适的人去排查和解决。

7.6 总结

在 CDN 的实际应用过程中，要把握好一些基本原则，这些原则对于在业务场景中恰当地使用 CDN 非常关键。笔者通过实践将 CDN 最重要的使用原则总结如下。

1) 尽早了解 CDN 的基本工作原理

无论是调度优化，还是命中率的调优，或是针对页面做的 CDN 静态化，了解 CDN 的基本工作原理都十分重要。非常清楚地了解 CDN 的优缺点和使用场合十分关键，要清楚地了解 CDN 能做什么、不能做什么，适合做什么、不适合做什么，以及使用之后要考虑什么问题。

2) 尽早了解 license 的细节，比较不同服务等级 license 的差别

CDN 提供商提供的 license 版本很多，需要和售前人员清晰地了解 license 提供的功能和 SLA 的细节，这样可以综合 ROI 多方面来确定使用哪个版本，也可以对某些高级功能进行测试和线上效果实测。

3) 做好上线前的测试验证工作

测试验证工作，需要有比较健全的功能验证流程。在沟通过程中，经常遇到各种口头“承诺”，双方在口径和理解上存在一定的误差。例如在 WebP 性能优化的项目中，首先笔者和售前人员沟通了方案的可行性，准备将 WebP 格式的图片上线替换 JPG 格式的图片，售前人员说这是可行的，即 CDN 提供商有能力缓存 WebP 格式的图片。上线测试时，我们发现性能并没有好转，一直猜测是由于上线新格式图片的范围太小，导致热度不够，热点不突出，最终导致命中率降低而引起的。后来去源站查看，发现 WebP 格式的图片大量回源，而且几乎每次访问都



产生了回源，再跟售前人员进行沟通，发现是配置没有启用。其实发生过多类似的问题，如前面提到的命中率超低的问题也是配置不正确引起的，这些问题都需要做好沟通和测试工作。

4) 尽早做好监控规划

细化监控对于发现问题非常关键，监控还需要做一定的规划，针对主要的域名最好做细分。针对不同类型的命中率需要分开监控，根据命中率从低到高的排行数据，能够对回源站的请求进行归类和分析。

5) 避免全量发布新图片格式

在使用 CDN 的过程中命中率越高对网站的卸载能力就越好，一般回源意味着性能会急剧下降，回源数量突增，非常可能造成系统故障。而全量发布新图片格式，会造成大量的回源，我们的网站已经因这个问题发生过多故障。例如在搜索列表页面，商品图片多达数十个，搜索访问量非常大，网站流量的很大比例来源于搜索，因此一旦有新图片格式，相当于在 CDN 节点上的缓存全部被击穿，回到源站，造成大规模的性能下降。对于这种情况，一般在发布时可以通过将发布周期变长来解决。例如某个应用有 100 台机器，根据源站能够支撑的容量，逐步切换机器来增加权重，同时观测监控系统的变化情况，关注关键指标是否发生变化。

6) 尽量对图片只新增不修改

图片一般很少会涉及修改，在设计时，尽量和图片名称解耦，这样可以做到图片只会新增而不用修改。图片新增设计对于提高图片的命中率将会起到非常重要的作用，涉及修改需求时，只需要修改图片的名称，也就是通过新增图片的方式来让图片过期，可以将图片的过期时间设置成 3 年，甚至 10 年（不能设置过长的过期时间，因为 CDN 的存储空间有限，会导致图片被淘汰、性能下降，特别是针对热度相近的场景）。

7) 尽早设计好源站架构

源站是高风险防范的最后一道防线，在大量回源时，能够抵挡住容量的压力，而在实际架构的过程中，特别是在网站的高速发展时期，在对 CDN 优化还不是很了解的时候，很容易出现 CDN 命中率不高的情况，源站承受很大的压力。当命中率不高的时候，静态资源的访问大量回源，CDN 的卸载能力没有充分发挥，这对源站的容量要求很高，特别是在网站发展初期，机房网络建设不健全的时候，交换机的千兆网卡，造成流量稍大时丢包大量增加，网络延迟大幅增加，用户无法及时打开网站，从而引起订单量大幅下降。所以在源站的架构优化过程中，要提前考虑网络架构的优化，以及源站吞吐量的优化。



8) 尽早了解 CDN 的配置规则

CDN 里面有些默认的配置规则,如默认用户真实 IP 地址的字段使用的是 TRUE_CLIENT_IP,默认没有配置的图片格式使用的是 WebP,这些配置是否开启需要 CDN 提供商进行准确的确认。

9) 提供合理、准确的数据对 CDN 进行优化

从前面的调度优化实例中可以看出,通过 LastMile 监控发现了非就近调度的情况,在提供了 Local DNS 的 IP 地址之后,问题得到了解决,并且免费升级了 license,使得调度上得到了非常大的优化。由此可见,提供准确的数据非常关键。

10) CDN 的性能问题一般和大量回源相关

由于某种原因造成大量的回源,会引起 CDN 性能的大幅下降。CDN 提供的 SLA 的基本保障是可用性,当性能发生衰减时,大都跟大量的回源相关,可以通过细分监控来确定是由哪个域名引起的,通过监控的衰变时间点,确定项目发布是否会引起大量的回源,一般成熟的 CDN 是能够保障稳定性的。

11) CDN 高命中率是 CDN 应用的基本要求

CDN 不仅能提高性能,也能提高稳定性,由于图片、JS、CSS 等静态资源访问量极大,一个页面请求可能达到数百个静态请求,源站的压力极大。在网站从小到大的发展过程中,由于机房建设的基础设施很难到位,把如此大的压力放到源站,源站的稳定性会受到极大的影响,所以高命中率是 CDN 应用的基本要求。从应用实践来看,CDN 的命中率对于用户类型的访问应该接近 100%,如果没有达到 100%,那么说明优化空间非常大。

12) 应用 CDN 需要考虑 ROI

CDN 的优化方案非常多,不是所有的 CDN 优化方案都适合在项目中使用,要把合适的方案应用在合适的地方。通过 ROI 分析,可以比较清楚地知道,哪些方案是有用的和合适的。CDN 的应用有时候会带来架构的复杂性,这些复杂性的引入不仅浪费大量的人力和物力,还会造成用户体验的下降。CDN 的应用一定要达到理想的目标,否则就是过度应用。



8

第 8 章

大型网站性能监控体系

大型网站的运行需要一个完善的监控体系，大型网站不同于小型网站，小型网站只需要数台服务器就可以提供服务，而大型网站则需要几千台到几万台服务器提供服务。当某个服务器出现问题时，没有完善的监控体系很难发现问题，而且大型网站的用户访问量在数量级上比小型网站大得多，一旦发生问题，对于业务的影响也非常大。因此对于大型网站而言，监控相对小型网站重要得多，例如对淘宝、天猫这种交易类型的网站，数分钟的不可用会造成数亿甚至数十亿元的经济损失，所以对网站的可用性、容量、监控维度、精细程度要求很高。监控体系是大型网站的咽喉所在，是一切问题分析、定位和解决的基础，监控的根本目的是发现问题并定位问题，是性能优化的基础。

本章主要讲述大型网站的监控需求和监控指标，以及如何实现监控，让我们一起揭开大型网站性能监控体系的面纱。



8.1 监控设计

监控的目的无疑是快速发现问题，并且能够通过监控的细分化来快速定位问题。监控在日常使用的过程中也会遇到很多问题，很多时候会出现该监控的没有监控到，该报警的没有报警，当发现问题时，却不能定位。本节将对这些问题进行总结，并且从应用的角度来阐述监控的思路，以便达到更快、更有效地监控。

8.1.1 应用监控存在的问题

在应用监控时主要存在如下问题：

- 监控缺失，关键问题发现不了，对关键指标缺乏监控。
- 对关键指标缺乏统一的梳理，对为什么要这个关键指标不清楚，不知道什么时候出现什么问题能够通过这个指标反映出来。
- 在设计时，以有无监控为基本要求，对监控缺乏“化整为零”的思路，对监控缺乏校验。
- 监控凌乱，缺乏整体的设计思路，想到哪做到哪，导致在问题排查时不知所措。
- 报警误报率高，骚扰业务。

8.1.2 从问题排查思路看监控的设计

从应用监控的角度来看，如前面所说，监控的目的是快速发现和定位问题，要想更快地发现和定位问题，一般分为两个大的步骤：

第一，要定位于问题发现，通过监控能够直接判断是否出了问题，是否对业务有影响，从这个角度来说，首先要考虑部署业务监控的能力，对业务的关键指标进行监控，同时要根据业务的周期性规律进行环比、同比的报警规则设置，出现对业务有直接影响的问题要立即报警。

第二，从定位问题的角度来看，要顺应排查的思路进行监控设计，当通过业务监控发现问题时，定位问题要选择对业务影响最大的问题。

首先排查应用是否产生了大量的异常，当业务有损伤时，会看是否是应用本身出现问题了。能够直接反映应用是否出现问题的指标是应用是否发生异常，所以异常是定位问题的直接指标，这里的异常包含代码运行过程中产生的异常。

其次看关键链路的调用次数是否变少了，调用次数变少的可能原因如下。



- 可能原因之一：外部流量减少。
 - 正常的业务影响：活动或者节假日影响。
 - 网络链路接入异常：如外部接入核心交换，通常多个运营商接入，某些运营商出现网络问题，这个问题一旦出现会导致整体流量下滑。
- 可能原因之二：链路耗时变长，导致在一定时间内处理的请求数变少了。
 - 交换机流量不均，某些机柜的流量被打满，TCP 重试，导致耗时变长。
 - 服务器流量不均，导致流量打满，TCP 重试，导致耗时变长。
 - 应用集群个体异常，某些服务器存在问题。
 - 集群系统指标个体异常，集群中个别服务器存在问题，可能配置不同（比如 CPU 核数不同）。
 - 依赖方个体异常，某些依赖存在问题，导致访问异常。

从上面的问题排查思路看，监控应该具备下列能力：

- 关键指标的监控发现能力。
- 对集群个体、整体、依赖方全方位的监控能力。
- 粗细结合的监控能力。粗粒度的监控主要为了快速发现问题，细粒度的监控能够快速定位问题。

8.1.3 监控的设计步骤

监控的根本目的是发现并定位问题，所以监控的体系化思路可以从发现问题到定位问题要做的工作出发考虑。监控的设计一般包含以下几个步骤。

1. 关键监控项梳理

梳理的目的是为发现和解决问题提供依据，通过关键监控的布点和报警来预知系统的问题，通过预设的问题预警来减少对业务的影响。通常监控项梳理主要包括业务关键指标，当业务关键指标出现问题时，一般都是大问题，所以在休息时间之外，只有业务关键指标出现问题才是需要跟进的。跟进还是不跟进从报警上需要进行分级，监控和报警分级是监控体系化的重要原则。报警是报给人的，报警过多，容易引起麻烦，报警缺失，发现不了关键问题，合理的分级能够在两者中取得平衡。

关键监控项梳理应该符合以下原则，这些原则的依据是排查问题的思路，当问题发生时，人们总会依据从发现问题到定位问题的思路进行梳理。



- 从重要到非重要排序

按照对业务的影响程度，对业务影响程度越大的监控项越应该排在前面。

- 从汇总到细分排序

汇总主要用于发现问题，细分主要用于定位问题。

监控角度分为两类，一类和发现问题相关，另一类和定位问题相关，业务出现问题，要么和业务本身相关，要么和系统相关。

对于应用监控来说，通常要包含如图 8-1 所示的关键监控项。

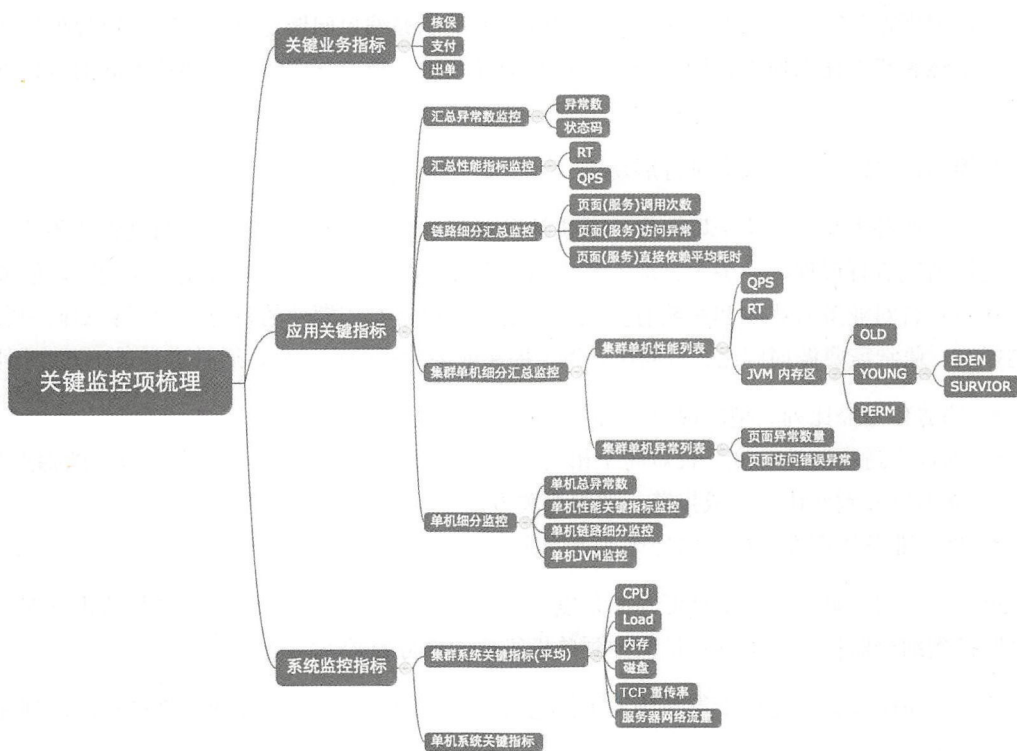


图 8-1

2. 关键监控项的报警抽离分级

- 临界点报警。
- 业务预警做重点关注报警，可以通过收集短信的方式进行预警。



3. 监控布点开发和设计

监控需要各种数据的支持，一般而言，布点要和关键指标进行关联。

- 捕获关键异常，对关键异常进行日志记录，并在记录时进行分级。
- 对直接依赖的运行耗时进行日志记录。
- 对反映业务指标的运行情况进行日志记录。
- 正确处理错误码。很多时候，我们会将错误的 HTTP 返回码处理成正确的响应，如 404 错误，如果在 Response 返回时没有设置，会被处理成 200 的状态码。

4. 监控展现设计

监控展现通常容易被人忽略，一个好的监控展现能够缩短问题定位的时间。监控展现要体现一目了然和集中化的原则，切忌分散，否则会导致排查根本问题时，需要从不同的入口逐个跟进。

监控通常只需要 3 个大盘进行展现。

第一，业务大盘，定位于以终为始地快速发现问题。业务大盘展现的目的是将业务关键指标的运转情况进行展现，业务大盘通常需要放在首要的展现位置，当业务发生问题时，能够快速地获知是否对业务有影响和影响有多大。由于业务通常有周期性的特点，为了获知问题发生时的影响，通常需要做周同比、月同比。为了提高业务大盘的可读性，通常遵循以下几个原则。

- 趋势化监控比列表型展现更一目了然。
- 确定合适的监控粒度，长短间距相结合。短粒度是为了确保当业务发生极大的拐点时，能够感知到变化，形成报警，通告相关方。
- 确定业务周期性规律，做环比或者同比。

第二，应用大盘，定位于从汇总的角度来看是否是应用本身的问题。顺序是从粗到细，从汇总监控到细分监控，从集群总量监控到单机维度的细分监控。

第三，系统大盘，定位于当系统发生问题时使用。同样，系统大盘应该遵循从汇总到细分监控的原则。

5. 调试和调整报警规则

实际上，在正常情况下业务也会发生抖动，监控预警需要将非正常抖动定位出来。通常报警规则有初始化预设的过程，通过一段时间的实践可将阈值进行逐步调整。阈值的设置要考虑如下因素。



- 活动引流。
- 节假日。
- 业务调整。
- 业务自然高峰和低峰期，在业务低峰期，同比和环比阈值变化特别大，所以在业务低峰期，预警可以不设置（当然和具体的业务相关）。

6. 监控设计和结果审核

对监控的设计和部署的结果进行审核是避免监控缺失的兜底手段，通常人们会认为，监控是可有可无的东西，但实际上监控是快速地发现 and 定位问题的重要手段。通过好的监控，可以大大降低当问题发生时对业务的影响。在日常监控的过程中，我们经常会遇到各种问题，总认为监控很简单，但是很多时候会碰到监控杂乱无章的现象，而且经常会发现，工程师部署某个监控的目的是发现和定位什么问题并不清楚，所以在监控的设计和实现过程中，监控结果的审核是必不可少的流程。

7. 预警演习

监控是需要人工主动关注才能发现和跟踪问题的，但是我们不可能不做别的工作，整天盯着监控大盘看，所以预警的作用就非常明显。预警及时，对业务的影响小，预警缺失，对业务的影响可能会非常大，但是我们并不能确保预警一定是正确的、预警的阈值一定是合理的。预警配置了，但是并不知道有没有生效。所以在实践的过程中，笔者将预警的演习作为监控的体系化流程设计中不可或缺的一环。

8.1.4 监控常见法则总结

- 对监控进行体系化设计。
- 梳理和部署关键指标时，能够有的放矢，对每项指标监控的目的要非常清楚，在发现问题或定位问题时能够通过监控这个指标达到目的。
- 对关键指标的预警进行演习。
- 对监控的部署结果进行审核，并且能够得到组织保障。
- 以终为始地发现问题，将能够直接影响业务的监控项进行优先部署，通常对关键的业务指标进行梳理和部署。
- 采用合理的时间间隔粒度，长短时间间距相结合，对每个间隔的业务影响能够一目了然，通常的监控时间间隔粒度包括小时监控、天监控和 5 分钟监控。
- 对于定位问题的应用和系统的关键指标进行监控。
- 完善监控埋点，对关键业务和流程进行日志记录，特别是异常。

- 调试报警规则，根据业务周期性规律设置报警规则，并且对报警阈值不断地进行调整。
- 对报警进行分级。
 - 按照业务高低峰时间段进行分级，在低峰、低业务量时调整对比时间。
 - 报警方式分级：对直接影响业务的监控项进行短信报警，对系统的问题进行 IM 报警，不仅可以节省成本，而且可以让工作更加高效。

8.2 大型网站性能监控体系设计目标和原则

大型网站对于问题的发现、诊断、解决的时间要求极高，一点时间的拖延，不仅会造成重大的损失，也会造成极差的用户口碑。网站的稳定性和性能建设对于一个稍大规模的网站来说，非常重要。大型网站监控一般遵循下列原则，而监控体系的构建和设计，也需要围绕以下几点进行。

8.2.1 准确性

准确性是监控的首要原则，准确性有两个含义。

其一是监控系统应该一直持续可用。试想如果在淘宝这样量级的网站组织一次促销活动，很可能因为监控系统出现宕机而损失惨重。当系统出现问题时，没有监控系统可用，什么时候出现问题，出现什么问题，以及问题的严重性，都一无所知，这对大型网站来说就是一种灾难。

其二是监控系统应该保持数据的正确性。一般监控系统因为在网站的后台，不是网站用户能够触及的地方，因而很容易被人忽视。在网站的构建过程中，有太多因为监控数据的不正确和不可用，造成分析问题极其困难。大量的监控系统由不同的人构建，大多数监控系统因为监控数据存在严重的问题而下线，监控系统从某种程度上说和网站面向用户流程的页面一样重要。

为保证正确性和可用性，监控系统需要纳入项目管理流程中，有需求收集、产品设计，也需要有测试流程，并且需要有一个专门的团队来开发这些系统，而不是一个开发人员利用业余时间开发监控系统。当业务需求来的时候，一般都需要优先保证业务需求，监控系统在这种投入和开发模式下不可能得到很好的对待和发展。说到底，监控系统是网站的咽喉，需要正确地对待和投入，需要从组织结构上得到保障。

8.2.2 完整性

完整性指的是，在诊断问题过程中，需要把几乎所有需要的监控项监控完整，这也是性能监控非常重要的一个原则。笔者曾经在排查一个性能问题时，因为运维人员启动了一个凌晨 3 点的定时任务，导致每天 3 点 CPU 占用率极高，出问题机器的因负载过高几乎不可用。试想如果没有操作系统定时任务数据的监控，这个问题几乎是不可能被发现的，或者很长时间才能排查出来。另外有一次在排查性能问题的过程中，由于核心路由器没有细粒度时间维度的监控导致排查容量问题用了 14 天时间。监控数据需要完整，监控项需要完整，没有完整的数据，排查问题的效率将大大降低，有时候甚至排查不了问题。

8.2.3 实时性

实时性对于监控系统非常重要，问题发生时需要及时发现，如果问题发生后，因为监控系统的架构设计问题造成发现问题延迟，会造成非常大的影响。数据的实时性要求是大型网站的基本要求，在现有的大数据工具日益成熟的情况下，构建实时性监控系统的难度变得越来越小。实时性是大型网站的基本要求，越实时，越能提早发现问题，一般一分钟的监控粒度是最基本的要求。这里面还有个平衡的问题，监控粒度越细，采集指令的运行越频繁，对系统的资源占用也越大，对应用本身的性能影响也越大。一分钟延迟，是大型网站最基本或者最低的要求。

8.2.4 细分化

监控的细分化对于排查系统的瓶颈、追踪系统问题是非常关键的，监控数据如果是粗粒度的，很难知道问题在哪里。监控的细分化对于所有的监控都是非常重要的，无论是对于排查业务问题，还是排查性能问题或其他问题，监控越细分越能找到问题的所在。例如 CPU 消耗的监控，CPU 消耗包含 User、System、IOWait、Nice、IRQ、SRQ 等消耗，如果只是给出 CPU 总体的消耗，而没有细分到这些维度上，很难知道 CPU 消耗在 IOWait 上，还是消耗在用户空间上，对于问题的判断会造成很大的难度。例如对订单的监控，如果不细分到某个地区、某类用户，就不能通过订单量的下降，得出是运营商的问题，还是程序本身的问题。

8.2.5 聚合化

大型网站往往有数千台服务器，一个大型的应用集群可能需要上千台机器，查看这些机器汇总的表现，非常重要。例如网络总体流量突增，突增的规模有多大，容量保障的时候，是否和预期的 QPS 一致，不可能把每台机器的 QPS 加起来一个一个算，对于 RT，我们不可能把每

台机器的 RT 单独汇总，一台一台地查看机器的表现。

8.2.6 图表化

图表化指的是将数字化的东西用图形的方式进行展示，人们通过图表很容易看到发生问题的时间点，例如订单的趋势监控，一般是以周同比、月同比的数据进行对比的。对于一个成熟的网站，订单的数量和下单时间段一般都是比较固定的。将每分钟订单连接成图线趋势进行多个图形对比，可以很容易地发现问题发生的时间点。图表化很容易表达出问题，方便人们直观地查看。

实际上，我们经常使用各种图表来定位问题，例如，看 RT 趋势图，辨别问题，一般来说，RT 变大或者突然变小是有问题的象征；QPS 趋势图如果上涨或者下降，要么有攻击，要么网站存在故障，造成用户访问量下降。通过图表可以直接定位问题，而不需要依靠数字化的抽象对比来发现问题。图表化将问题的定位变得更加高效。

8.2.7 可追溯

大型网站一般代表着更多的服务器，规模越大的集群，出现问题的类型也会越多，有些问题转瞬即逝，当问题发生后，模拟和重现是非常困难的，因此监控结果需要保留一段时间。在条件允许的情况下，尽量将监控聚合的结果保存一定的时间，由于用户访问具有周期性的规律，通常监控结果需要保留一年左右，以便进行同比、环比等。通过对比，更容易发现和定位问题。保留监控结果，为可追溯提供了基本条件。问题可以追溯和跟踪非常重要，因为问题的重现需要耗费大量的人力和物力，同时模拟一模一样的环境也是非常困难的。

注：一般监控并不都是秒级的监控，业务量越大监控的时间粒度越小，通常大型网站一般会采用 5 分钟的监控粒度，访问量更大的系统通常是分钟级别的监控，对于超大型网站，如果条件允许最好能实现秒级监控。

8.3 性能指标和监控项及实现

通常性能问题和很多因素相关，例如访问量突增、某个定时任务启动、磁盘清理、机器老化、耗费资源的特殊业务执行等。大型网站的监控体系一般都是通过采集日志的方式，通过大型分布式日志系统，如 Jstorm、Hadoop 集群，将日志进行分析计算，最后通过前台 Web 系统将监控结果展示出来。监控的构建包含日志埋点、日志采集、结果计算和结果存储。

大型网站监控构建有以下几个目的。

1) 性能容量度量

做一个大型网站，需要很多机器，要提前进行机器的采购，采购机器的数量可以根据单机性能容量进行评估。

2) 问题预警

监控是预警的前提条件，挑选出对业务有影响的监控指标，进行问题预警以便减少对业务的影响。

3) 问题追溯和分析

对关键监控项进行监控并且将监控数据沉淀下来，就可以很方便地进行问题分析。大型网站需要一种比较高效的定位问题的方式和工具，在数百台服务器中发现并跟踪问题，没有工具的支持是非常难的，提前对监控进行规划和部署，不仅能够快速地定位问题，而且能够通过监控到的问题指导架构的升级。

4) 高效的运维管理

运维体系需要清楚地知道集群有多少台机器，可以批量进行上下线操作，可以批量地对机器进行重启、磁盘扫描和清理，同时通过大量工具的支持，让运维和业务紧密结合，以便发挥监控的最大作用。

从用户访问链路出发，整个性能监控体系分为：

(1) 静态资源访问的性能，前端页面性能监控。

(2) 机房访问监控。

从性能监控的角度来说，从机房整体出发，可以分为 Web 集群监控、中间件监控、单机监控、网络设备（交换机、路由器）监控、负载均衡设备监控。

从部署的视角来看网站的部署架构如图 8-2 所示，要快速地部署监控以便发现性能问题，需要对这些物理的监控项进行监控，并且实现力所能及的可视化和工具化，这是大型网站监控的基本要求。

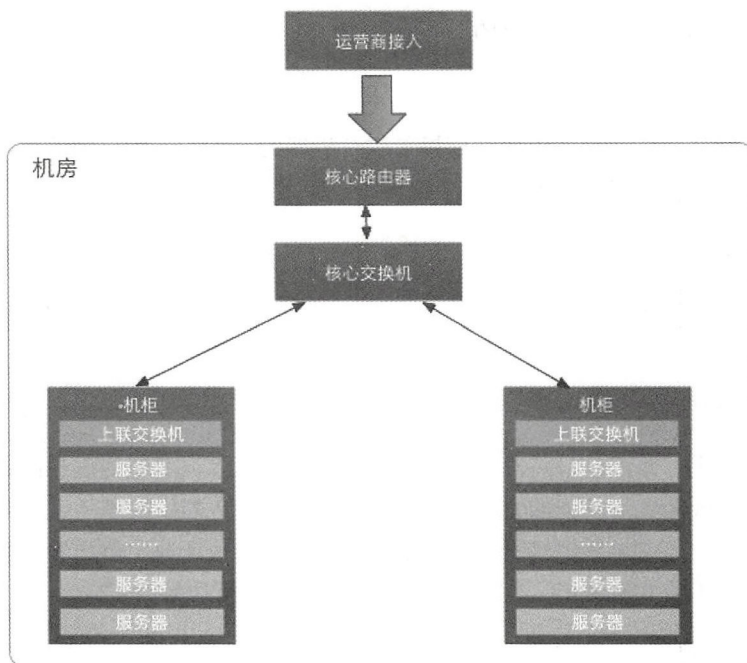


图 8-2

1. 机房部署结构

• 运营商接入

一般大型网站都会有多个运营商接入，多个运营商接入主要是出于可用性和性能考虑。使用相同运营商的接入性能更好，使用多个运营商的接入满足不同的用户对不同运营商接入的快速网络访问要求，多个运营商接入可以避免单运营商网络不可用带来的影响。笔者曾亲身经历过一次由于运营商调整网络配置导致网络不可用，影响了部分地区用户的访问，当然这需要很强大的监控体系。机房内部监控主要覆盖请求已经到达机房的情形，但是终端用户网络不可用，同时由于用户设备到机房的链路是断的，监控数据也无法从客户端采集到服务器端，因此需要借助第三方的力量解决问题，例如手持终端的 Google Analysis 监控，监控脱离于网站自身的机房。如果要借助 LastMile 监控则需要部署足够多的样本去测试。

• 核心路由器

路由器位于 OSI 7 层模型中的网络层（第 3 层），主要负责将用户请求的数据包路由到目标地址，同时将服务器响应的数据包回应给用户端。在机房内部，路由的网络构建可以由 RIP

或 OSPF 网络来进行。

- 核心交换机

交换机一般工作在数据链路层，识别目标机器的 MAC 地址完成数据包的转换。所有跨机柜服务之间的调用都需要经过核心交换机。由于交换机工作在第二层，转发速度极快，有效地解决了网络拥堵问题。

- 机柜

机柜主要用来存放服务器，标准机柜尺寸有很多种，其中 42U 就是标准机柜尺寸之一。U 是国际上通用的机柜内部尺寸的特殊计量单位，U 指的是机柜内部的有效使用空间。因此，U 就是衡量标准机柜尺寸的一种符号。标准机柜尺寸也分为很多种，其中有 37U 机柜、47U 机柜、20U 机柜、32U 机柜、6U 机柜、12U 机柜等。不同规格的机柜存放的服务器数量不同。常用的有机架式服务器和刀片式机柜，刀片式机柜能够放置更多的服务器，但是散热性能比较差。刀片式机柜有几个不同的机框。机柜通常有上联交换机，用于和核心交换机进行数据传输和交换。通常在设计监控体系时，需要监控机柜的网络流量，大型网站演变过程中的网络瓶颈是常见的一种瓶颈，一个机柜的网络带宽是有上限的。最早的时候，笔者所在网站曾经使用刀片式机柜，一个机柜存在多个机框，每个机框是 1024MB（1GB）网络，一个机框放置 16 个服务器，收敛比是 1：16，每个服务器的带宽上限平均为 64MB。对于大型网站，通常还大量用到 1 虚 4，一个服务器虚拟出 4 个实例来提供服务，这样平均分配到每个虚拟机上的带宽就更少。

- 服务器

服务器监控通常是为了定位服务本身的性能瓶颈，包括系统目前的水位，分为应用监控和系统监控。从上面的机房结构可以知道，为了定位问题，不仅要建立基于结果的监控，还要建立基于链路的监控，以便快速地分析问题。

2. 监控指标梳理

要定位问题和分析问题，从机房的结构上看，需要监控下列关键指标。

- 核心路由器监控：入口流量，出口流量。
- 核心交换机监控：入口流量，出口流量。
- 服务器监控：CPU，负载，内存，磁盘 I/O，网卡网络 I/O，TCP 连接数，线程数。
- 应用监控：（集群）应用 QPS，RT，链路分段 RT，异常，应用 URL RT，应用 URL 错误码，JVM。

8.4 性能监控的关键指标

性能优化首先要定位出性能瓶颈，确定优化方向，才能做到有据可依，而完备的性能监控指标是高效定位和分析瓶颈的前提。一般而言，要定位出问题，先看资源消耗情况，定位出哪些监控指标资源消耗多，然后看细分项，看看哪个业务消耗资源多，逐步细分下去，最终找到系统的瓶颈。

要精确定位性能的瓶颈并不是简单的事情，服务器应用构成（内核、应用、系统）十分复杂，要从众多的监控指标中定位出瓶颈，需要十分完善地诊断监控指标。下面将按照系统的调用层次和调用链路进行分类。

8.4.1 应用监控

应用监控指的是对部署在服务器上的应用本身的各种运行状况和资源的使用情况进行监控。应用监控主要聚焦在应用本身上，通常包括 QPS 监控、RT 监控、URL 监控（某个页面）、页面访问链路监控和页面访问状态码监控。

1. QPS

QPS（Query per Second）表示每秒处理完成的请求数，是衡量系统吞吐量大小或者系统处理能力的指标，服务器的 QPS 越高，提供服务需要的机器就越少。每个系统都有自己的峰值 QPS，表示系统最大的吞吐量。在做容量规划时，峰值 QPS 作为容量规划的一个主要因子，来计算需要多少台机器。可以根据一定的数学计算得出业务访问需要多少 QPS，再除以 Peak QPS，就可以得出需要机器的数量。

注：QPS 和 TPS 是有区别的，TPS 是每秒处理完成事务的过程数，一个请求可能包含多个事务处理过程，对于 Web 请求而言，一个单独的请求算作一个 Query，所以通常 QPS 是衡量服务器性能的主要指标。

- 对于 Web 系统而言，QPS 监控大部分都是采用 HTTP 来实现的，可以将日志记下来，一般称为 Cookie log，对于常见的 Web 服务器，如 Apache、Nginx，都可以配置访问日志，对于 Nginx：

$$\text{QPS} = 1 \text{ 秒的日志数量}$$

```
log_format accesslog '$remote_addr - remote_user[time_local] "$request" '
'$status $body_bytes_sent "$http_referer" ' '"$http_user_agent" $http_x_
forwarded_for';
```

- 通常 QPS 监控最好能够区分正确的请求和异常请求，正确的请求包含的 HTTP 状态码是 200，异常请求包含 404、302 和 500 错误请求。通常在容量测算时，需要将异常请求控制在一定范围内。例如，在大促之前，通常会根据业务量来评估。

2. RT

RT (Response Time) 表示响应时间，它是从接收请求开始到服务器处理完成的时间差值。RT 的长短取决于应用的特点，对于大型网站而言，所有应用都通过 RPC（远程接口调用）服务化的接口得到需要的数据，大部分时间消耗都在网络和远程接口的处理上。减少 RT 的方法比较简单，要么合并请求，将多次调用合并成一次调用，要么减少调用。减少远程接口调用，可以通过缓存架构升级或者去除多余调用来实现。应用本身的时间大部分都依赖于 CPU 的处理时间，CPU 的处理时间一般较短，当然应用本身仍然有很多瓶颈，例如锁等待和线程池连接等待。

对于大型网站而言，由于需要大量的服务器资源，其性能优化的目标通常是针对 QPS 来进行的，大型网站通常跨地区、跨国家，甚至跨洲，网络时间消耗占据 90% 的总时间消耗，所以优化服务器 QPS 目的是提升处理能力。优化 RT 通常不会在服务器上下工夫，通常以浏览器端（减少 HTTP 请求、预加载、延迟加载、异步化）和网络处理架构（如 DNS、TCP、CDN）的优化作为主要方案。

3. 并发数

并发数是衡量系统压力的指标，并发数越大系统压力越大。并发数是一个比较笼统的概念，通常在性能测试中使用。做性能测试时，为了测量系统的峰值 QPS，需要给系统一定的压力，而并发数的调整是系统加压的一种方法。一般并发数是针对某个具体的链路而言的，并发数是同时在处理或者需要处理的任务数或者线程数。

以上的三个指标是最终系统运行过程中性能的表现，数据的高低是各种资源综合作用的结果，与应用本身程序编写的问题、应用本身的特点（如 CPU 消耗大）和应用本身的架构（如计算量大、无结果缓存或者无法进行结果缓存、网络传输量大、网络瓶颈）都密切相关。

4. 异常监控

通常在服务器端出现峰值时会出现一些异常，这些异常主要是应用本身的异常，最常见的异常包括 RPC 连接池异常、数据库连接池异常、超时异常、JVM 异常。通常在单机或者集群进行压力测试时，异常作为停止的指标。同时，异常也是发生问题时首先要关注的，它能够帮

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

我们快速地定位问题发生的具体位置。

5. URL 监控

URL 是 Web 应用通常需要监控的，包含以下关键指标。

- 调用量：调用次数及趋势监控，可以进行同比、环比。
- 耗时：访问此 URL 的服务器端耗时。
- 错误数：在选定的周期内，访问此 URL 发生的错误数，最好将错误的具体信息进行聚类统计。分析问题，一般看到同类错误信息多，就可以判定是哪个调用出现问题了。
- 相关依赖明细：此 URL 包含的服务、存储等直接依赖的调用耗时，有了该明细便于定位出哪个服务出现了问题。
- 状态码：状态码监控能够快速获知正确的处理情况，如服务器处理错误情况、跳转情况等。大量的性能问题往往伴随着大量的错误处理，例如大量的 404 会导致 RT 急剧下降，因为没有完整地走完正常的业务处理流程。
- 并发数：服务器同时处理该 URL 请求的数量。

6. 关键方法监控

如果此应用不是 Web 应用，只是一个服务或者中间件，那么需要监控以下指标。

- 耗时：关键方法的耗时。
- 依赖明细：关键方法依赖的服务方法的耗时。
- 异常：调用出现的异常统计，包括异常的分类统计，用于定位和分析问题。
- 并发数：服务器同时处理该方法请求的数量，一般进入该方法体开始计数。

7. JVM 监控

- 分区监控：Eden、Survivor、Old、Perm Space 的大小变化。
- 垃圾回收次数：在监控窗口内 Young (Minor) GC 和 Old (Major) GC 的垃圾回收次数。
- 垃圾回收时间：在监控窗口内 Young (Minor) GC 和 Old (Major) GC 的垃圾回收时间，垃圾回收时间长说明 JVM 需要优化。

8.4.2 系统监控

系统监控指的是服务器操作系统相关性能指标的监控，通常包括服务器的 CPU 资源监控、网络监控、内存监控和磁盘监控。



1. CPU 资源监控

CPU 对于服务器端的性能起着非常重要的作用，CPU 是操作系统的调度中枢，是影响服务器端吞吐量的最重要因素。服务器峰值的处理能力取决于 CPU 的利用率和 CPU 的处理时间。所以监控 CPU 的使用情况，能够获知系统瓶颈的变化，以及在发现问题时是否可以通过水平扩展机器数来解决问题。如果一个系统或者应用过载，CPU 的利用率很高，CPU 处理时间很长，说明系统的瓶颈在 CPU 上，这时可以通过一些方式优化 CPU 的使用来提升系统的吞吐量。

- CPU 利用率：指的是进程中 CPU 消耗的时间。在 Linux 内核的操作系统中，进程是根据虚拟运行时间（由进程优先级、nice 值加上实际占用的 CPU 时间进行动态计算得出）进行动态调度的。在执行进程时，需要从用户态转换到内核态，用户空间不能直接操作内核空间的函数。通常要利用系统调用来完成进程调度，而用户空间到内核空间的转换通常是通过软中断来完成的。例如要进行磁盘操作，用户态需要通过系统调用内核的磁盘操作指令，所以 CPU 消耗的时间被切分成用户态 CPU 消耗、系统（内核）CPU 消耗，以及磁盘操作 CPU 消耗。执行进程时，需要经过一系列的操作，进程首先在用户态执行，在执行过程中会进行进程优先级的调整（nice），通过系统调用到内核，再通过内核调用，硬中断、软中断，让硬件执行任务。执行完成之后，再从内核态返回给系统调用，最后系统调用将结果返回给用户态的进程。
- CPU us 利用率：进程在用户空间消耗的 CPU 占比，通常指的是一个监控周期内应用进程的 CPU 消耗的占比。
- CPU sy 利用率：进程在内核空间消耗的 CPU 占比，称为 PRI。PRI 是比较好理解的，即进程的优先级，或者通俗点说就是程序被 CPU 执行的先后顺序，此值越小进程的优先级别越高。
- NI：就是我们所说的 nice 值，它表示进程可被执行的优先级的修正数值。PRI 值越小越快被执行，加入 nice 值后，将会使得 PRI 变为 $PRI(new) = PRI(old) + nice$ 。由此看出，PRI 是根据 nice 排序的，nice 越小 PRI 越靠前（小，优先权更大）则越快被执行。如果 nice 相同，则进程 UID 是 root 的优先权更大。在 Linux 系统中，nice 值的范围从 -20 到 +19（不同系统的值范围是不一样的），正值表示低优先级，负值表示高优先级，值为零则表示不会调整该进程的优先级。具有最高优先级的程序，nice 值最低，所以在 Linux 系统中，nice 值为 -20 表示一项任务非常重要；与之相反，如果任务的 nice 值为 +19，则表示它是一个高尚的、无私的任务，允许所有其他任务比自己享有更大使用份额的 CPU 时间，这也就是 nice 名称的由来。
- ID：进程在计算 CPU 消耗时，如果 CPU 处于闲置状态，那么记入闲置 CPU，即 ID。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

- **WA**：进程在等待磁盘时的 CPU 消耗，这个值越大，说明磁盘处于越繁忙的状态。
- **HI**：硬中断，指的是和系统相连的外部设备（如网卡、硬盘、USB、键盘等）自动产生的，主要用于通知操作系统外设状态的变化，如网卡收到一个数据包，就会发出一个中断。硬中断 CPU 使用占比指的是用于处理硬件中断的 CPU 消耗的占比。
- **SI**：软中断，在 CPU 获取硬中断时，CPU 把结果放在寄存器中，操作系统内核会启动 ksofttrip 进程来获取操作结果，这个过程叫作软中断。软中断值高通常是由网络设备引起的。
- **ST**：一般服务器部署都会使用虚拟技术，一个物理机会虚拟出多个虚拟机，当消耗 CPU 的应用和其他应用部署在一个物理机上时，会发生窃取现象。如果发生这种问题，可以调整物理部署，尽量将同类型的应用部署在一个物理机上，以确保流量相对均衡。
- **CPU Load**：包括 Load1、Load5、Load15，指在特定的时间间隔内运行队列中的平均进程数。

在 CPU 监控中需要特别注意下列问题。

- **Runnable 状态不等于 Running 状态**

处于 Running 状态的进程数量始终会小于等于 CPU 的核数，但是 CPU Load 指的是 Runnable 状态的进程数，这就是我们可以看到 CPU Load 的数量远大于 CPU 核数的原因。所有的不被调用 sleep、wait、stop、yield 方法的进程都是 Runnable 状态的，这些任务都在基于红黑树数据结构的可执行队列上。

- **UNINTERRUPTIBLE 任务被计入 CPU Load**

UNINTERRUPTIBLE 任务在 Linux 内核实现过程中仍然放置在红黑树里面，不会被调度到等待队列，把它计入 CPU Load。TASK_UNINTERRUPTIBLE 状态存在的意义在于，进程对某些硬件进行操作时（比如进程调用 read 系统调用，对某个设备文件进行读操作，而 read 系统调用最终执行到对应设备驱动的代码，并与对应的物理设备进行交互），可能需要使用 TASK_UNINTERRUPTIBLE 状态对进程进行保护，以避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。这种情况下的 TASK_UNINTERRUPTIBLE 状态总是非常短暂的，通过 ps 命令基本上不可能捕获到。TASK_UNINTERRUPTIBLE 在大量读取磁盘的时候又总能够轻易被捕获到，这类进程 kill -9 杀不掉，而且可以肉眼轻易看出它被累加在 Load 的值里面。

- **网络等待的进程（或者任务）在等待期间不消耗 CPU 资源**

代码调用 wait 或者 sleep 方法，CPU 资源将被释放，远程调用等待服务器响应期间，客户



端的 CPU 资源不会被消耗，所以在优化远程调用时间时，如果 CPU 是瓶颈，则优化 RT 对提升峰值 QPS 并没有帮助。

CPU 监控的实现内核中有个系统定时器周期性地更新系统运行时间（用 `cat kernel/.config | grep '^CONFIG_HZ='` 命令查看），更新实际时间，检查当前进程是否用尽了自己的时间片，更新资源消耗和处理器时间的统计值。系统定时器以固定频率产生中断，越高频率的时钟中断意味着越精确的进程抢占，越准确地执行依赖定时值执行的系统调用（如 `poll` 和 `select`）。系统定时器的每次中断称为 tick，如果内核时钟的频率是 100HZ，那么每秒的 tick 为 100，每 10ms 时钟中断一次。每次 tick，如果 CPU 正在执行用户态的进程，那么用户态的 CPU 消耗加 1，如果 CPU 正执行内核态的进程，那么 system 的 CPU 消耗加 1，如果 CPU 在等待磁盘响应，那么 `IOWait` 的 CPU 消耗加 1，如果 CPU 处于闲置状态，那么 `Idle` 的消耗加 1。

在网络等待期间，进程被 Linux 内核调度出可执行进程队列，放入等待队列，因此远程等待的任务不会消耗 CPU 资源，所以在进行服务器性能优化时，如果系统瓶颈是 CPU 时间，那么减少远程调用的时间不会提升 QPS。

2. 网络监控

网络监控通常指的是服务器网络监控，用于定位服务器网络瓶颈，通常看 TCP 的重传率，内部网络重传率超过 1%，往往会造成访问延迟增加。网络瓶颈的重要标志是网络重传率变大，例如 1000MB 带宽上限的服务器，当流量超过 800MB 时，网络重传率会有所增加，此时是明显的网络瓶颈。

网络监控包含以下关键指标。

- **TCPretr**: 网络重传率，可以通过统计一段时间内发出去的包的个数和重传包的个数来计算，在 Linux 操作系统中可通过多次查看 `cat`、`proc`、`net`、`snmp`，计算差值来进行统计。造成网络重传的原因很多，包括存在于路由硬件的各种网络风暴、交换机和服务器的带宽，但是服务器响应慢（除非 CPU 资源耗尽）不会引起网络重传。TCP 是双工协议，请求发送端将请求报文发送给服务器端，服务器端接收完整的请求，报文才开始进行逻辑处理。服务器端处理完请求后，会给客户端发报文，重传率是发起端统计的，从这个过程看，重传不会发生在服务器响应时间慢的情况下，除非服务器端的 CPU 耗尽，导致没有资源接收处理客户端发送的请求报文。
- **网卡 I/O**: 网卡的进/出流量，用于监控网络带宽的使用量，使用量超过阈值通常会造成明显的网络瓶颈。



- TCP 连接数：TCP Established、TCP Time_wait、Closed、Idle，需要重点关注 timed_wait，如果 timed_wait 值过高，说明在连接处理上存在问题，需要优化。

3. 内存监控

系统的内存不足引发磁盘交换，很可能使内存成为“木桶短板”，内存监控通常包含以下关键过程指标。

- 内存的使用量：物理内存的使用量。
- swap I/O：交换到磁盘的量，常常是因为内存不够才发生的。

4. 磁盘监控

磁盘 I/O 性能监控指标如下。

- 磁盘利用率（Utilization）

磁盘利用率表示磁盘处于活动时间的百分比，磁盘在数据传输和处理命令时处于活动状态。如果磁盘利用率超过 70%，应用进程将花费较长的时间等待 I/O 完成，因为绝大多数进程在等待的过程中将被阻塞或休眠。磁盘利用率越高，资源争用就越严重，性能也就越差，响应时间就越长。

- 服务时间（Service Time）

服务时间包括寻道、旋转时延和数据传输等时间，即磁盘读或写操作执行的时间。其大小一般和磁盘性能有关，CPU 和内存的负荷也会对其有影响，请求过多也会间接导致服务时间增加。如果服务时间持续增加，可能是磁盘瓶颈问题。

- I/O 等待队列长度（Queue Length）

I/O 等待队列长度指待处理的 I/O 请求的数目。对于磁盘阵列部署而成的逻辑驱动器，需要获取实际物理磁盘数目，以获得平均单块硬盘的 I/O 等待队列长度。如果 I/O 请求压力持续超出磁盘的处理能力，该值将增加。

- 等待时间（Wait Time）

等待时间指磁盘读或写操作等待执行的时间，即在队列中排队的时间。如果 I/O 请求持续超出磁盘处理能力，意味着来不及处理的 I/O 请求不得不在队列中等待较长时间。



通过监控以上指标，并将这些指标的数值与历史数据、经验数据及磁盘标称值进行对比，必要时结合 CPU、内存、交换分区的使用状况，不难发现磁盘 I/O 的潜在或已经出现的问题。还有几个其他指标，对识别瓶颈的作用不是特别明显，如 IOPS 和 TPS，不详细解释了。

8.5 常用监控命令详解

常用监控命令详解如表 8-1 所示。

表 8-1 常用监控命令详解

| 类 别 | 监 控 命 令 | 描 述 | 备 注 |
|-------|--|--|--|
| 内存瓶颈 | free | 查看内存使用情况 | |
| | vmstat 3（间隔时间）1000（监控次数） | 查看swap in/out详细定位是否存在内存瓶颈 | 推荐使用 |
| | sar -r 3（间隔时间） | 和free命令类似，查看内存的使用情况，但是不包含swap的情况 | |
| CPU瓶颈 | top -H | 按照CPU消耗的高低进行线程排序 | |
| | ps -Lp \$进程号 cu | 查看某个进程的线程CPU消耗排序 | |
| | cat /proc/CPUinfo grep 'processor' wc -l | 查看CPU核数 | |
| | top | 查看CPU总体消耗，包括分项消耗，如User、System、Idle、nice等 | |
| | top然后Shift + H：显示Java线程；Shift + M，按照内存使用进行排序；Shift + P，按照CPU时间排序；Shift + T，按照CPU累计使用时间排序。多核CPU，进入top视图按“1” | 专项性能排查，多核CPU主要看CPU各个内核负载是否均衡 | |
| | sar -u 3（间隔时间） | 查看CPU总体消耗占比 | |
| | sar -q | 查看CPU Load | |
| | tsar -CPU -l 1 -d 20150408 | 查看CPU消耗历史，一分钟一次 | 当回溯历史问题时，需要查看CPU是在系统上还是应用上，是steal消耗，还是nice消耗 |
| | top -b -n 1 awk '{if (NR <= 7) print; else if (\$8 == "D") {print; count++;} } END {print "Total status D: 'count'}' | 计算在CPU Load里面的uninterruptedsleep的任务数量 | uninterruptedsleep的任务会被计入CPU Load，如磁盘堵塞 |



续表

| 类 别 | 监 控 命 令 | 描 述 | 备 注 |
|------|--|-------------------------------|--|
| 网络瓶颈 | cat /var/log/messages | 查看内核日志，查看是否丢包 | |
| | watch more /proc/net/dev | 用于定位丢包、错包情况，以便识别网络瓶颈 | 重点关注丢包和网络包传送的总量，不要超过上限 |
| | sar -n SOCK | 查看网络流量 | |
| | netstat -na grep ESTABLISHED wc -l | 查看TCP连接成功状态的数量 | 此命令特别消耗CPU，不适合进行长时间监控和数据收集 |
| | netstat -na awk '{print \$6}' sort uniq -c sort -nr | 查看TCP各个状态的数量 | |
| | netstat -i | 查看网络错误 | 功能等同于netstat，但是比netstat更高效 |
| | ss state ESTABLISHED wc -l | 统计并发连接数 | 更高效地统计TCP连接状态为ESTABLISHED的数量 |
| | tsar TCP | 显示TCP重传率，Retran表示重传率，是网络瓶颈的标识 | 只能在阿里系的服务商中使用 |
| | cat /proc/net/snmp | 查看和分析240s内网络包量、流量、错包、丢包 | 用于计算重传率，TCPPrtr=RetransSegs/OutSegs |
| | ping \$ip | 测试网络性能 | |
| | tracert \$ip | 查看路由经过的地址 | 常用于统计网络在各个路由区段的耗时 |
| | dig \$域名 | 查看域名解析地址 | |
| 磁盘瓶颈 | dmesg | 查看系统内核日志 | |
| | iostat -x -k -d 1 | 详细列出磁盘的读写情况 | 当看到I/O等待时间所占CPU时间的比例很高时，首先要检查的就是机器是否正在大量使用交换空间，同时关注IOWait的CPU占比，如果大，说明磁盘存在大的瓶颈，同时关注AWait，AWait表示磁盘的响应时间，它应该小于5ms |
| | iostat -x | 查看系统各个磁盘的读写性能 | 重点关注AWait和IOWait的CPU占比 |
| | iotop | 查看哪个进程在大量读取I/O | 一般先通过iostat来查看是否存在I/O瓶颈，再定位哪个进程在大量读取I/O |



续表

| 类 别 | 监 控 命 令 | 描 述 | 备 注 |
|------|---|---|--------------------------------|
| | df -hl | 查看磁盘剩余空间 | |
| | du -sh | 查看磁盘使用了多少空间 | |
| 应用瓶颈 | ps -ef grep java | 查看某个进程的ID | |
| | ps -ef grep httpd wc -l | 查看特定进程的数量 | |
| | cat .log grep *Exception wc -l | 统计日志文件中包含特定异常的数量 | |
| | jstack -l \$pid | 查看线程是否存在死锁 | |
| | awk '{print \$8}' 2015-05-21-access_log grep '301 302' wc -l | 统计日志中301、302状态码的行数， \$8表示第8列是状态码，可以按实际情况更改 | 常用于应用故障的定位 |
| | grep 'productDetailNew' cookie_log awk '{if(\$10=="200"){print}}' awk '{print \$12}' more | 打印包含特定数据的12列数据 | |
| | grep "2013:07:07" cookie_log awk '{(\$12 > 0.3){print \$12 "—" \$8}}' sort > /home/tm/request_timer_0.7 | 对Apache或者Nginx访问日志进行响应时间排序，\$12表示Cookie log中的第12列，0.3表示响应时间 | 用于排查是否由于某些访问超长造成整体的RT变长 |
| | grep -v 'HTTP/1.1" 200' | 取出非200响应码的URL | |
| | pgm -A -f \$应用集群名称 "grep "" 301 ' /home/logs/.../2/05/21-access_log wc -l" | 查看整个集群的日志中301状态码的数量 | |
| | ps -ef grep [PID] wc -l | 查看某个进程创建的线程数 | |
| | find / -name '*监控体系.md' | 查找磁盘下所有的包含特定文件名的文件 | |
| | find / -type f -name "*.log" xargs grep "ERROR" | 统计所有的日志文件中包含ERROR字符的行 | 这个在排查问题过程中比较有用 |
| | jstat -gc \$pid | 查看GC的信息 | |
| | jstat -gcnew \$pid | 查看young区的内存使用情况，包括MTT（达到最大交互次数就被交换到old区），MTT是目前已经交换的次数 | |
| | jstat -gcold | 查看old区的使用情况 | |
| | jmap -J-d64 -dump:format=b, file=dump.bin PID | dump出内存快照 | -J-d64用于防止jmap导致虚拟机崩溃（JDK6有漏洞） |
| | -XX:+HeapDumpOnOutOfMemoryError | 在Java启动时加入，当出现内存溢出时，存储内存快照 | |



续表

| 类 别 | 监 控 命 令 | 描 述 | 备 注 |
|-----|--|--------------------|---|
| | jmap -histo \$pid | 按照对象内存大小排序 | 注意会导致full GC |
| | gccore \$pid | 导出完整的内存快照 | 通常和jmap -permstat/opt/**/ java gcore.bin一起使用，将core dump转换成heap dump |
| | -XX:HeapDumpPath=/home/tomy/logs -Xloggc:/home/ admin/logs/gc.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps | 在Java启动参数中加入打印GC日志 | |
| | -server -Xms4000m -Xmx4000m-Xmn1500m -Xss256k -XX:PermSize=340m -XX:MaxPermSize=340m -XX:+UseConcMarkSweepGC | 调整JVM堆大小 | XSS是栈大小 |



9

第 9 章

大型网站容量评估

9.1 容量评估概述

容量评估是指通过一定的测量、推理方法来确定网站的容量现状，为运维预算、运维设备采购和解决系统瓶颈问题进行预警，并最终作为系统的伸缩性和水平扩展性的依据。简单来说，容量评估解决的是机器扩容、什么时候扩容的问题。

容量评估是对网站目前能够提供的容量上限的清晰了解，是对机器的容量进行相对准确、有根据的评估的过程。在大型网站的运营过程中，会经常进行各种平台组织的促销活动，如阿里的双 11 活动。一旦进行活动，一般都需要经过比较合理的评估以确定需要扩容多少台机器，才能满足容量的需求。一般的容量评估需要经过两个过程：

- 单机峰值 QPS 的评估（单机峰值吞吐量）。

- 评估每个集群需要的容量。包括每个应用、中间件需要提供的峰值 QPS（吞吐能力），通过订单、PV 和 UV 等关键数据推导出各个应用、中间件、数据库和存储所需要的容量。

一般而言，容量评估的过程有以下特点。

1. 峰值保障

容量评估是为了保障峰值时用户访问的稳定性和可用性，峰值保障策略并不是以异常点尖峰时刻的值作为评估的结果，峰值通常指的是维持时间较长并且无抖动时的高点。

2. 去噪保障

大促容量保障是为了保证合理的用户访问的可用性和稳定性，并不是为了保障攻击类的容量需求，所以要有去噪保障。攻击类的需求往往在安全架构层面进行保障，所以大促期间可能发生 SYN Flood 攻击、XSS 攻击、CC 攻击等，这需要在大型网站准入控制上做很多工作，例如 4 层防攻击、7 层防攻击等准备工作和演习。

9.2 容量评估的特点

容量评估是人为推算结果，存在以下特点。

1. 合理性

容量评估其实很难保证精确性，所以在日常评估的过程中，基本要求是合理并且有据可依。精确性很难在容量评估中进行衡量。

2. 业务相关性

业务相关性指的是被评估业务的类型之间存在某种关系，例如对于一些冷门的行业，有稀缺性，因此在容量评估时，可以根据业务特性进行分流，二八原则适用。

9.3 单机峰值 QPS 的测算

单机峰值 QPS 是计算机器数的一个重要因子，其大小体现了单机吞吐能力，如图 9-1 所示。

总体单机吞吐能力的测算过程是通过逐步压力测试来完成的，当出现资源瓶颈时，压力测试停止，峰值 QPS 就是出现资源瓶颈时的 QPS。

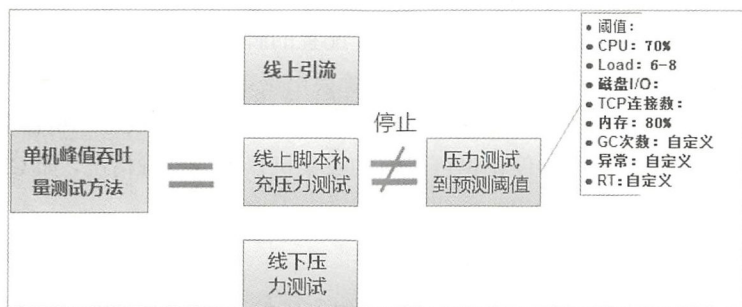


图 9-1

9.3.1 单机测算方法

单机测算方法如下。

1. 线下压力测试

压力测试到对应的资源瓶颈，通常使用 Jmeter 工具对应用访问的 URL 逐步加压进行测试，当超过阈值时可以停止，可以根据经验进行测试，以便提高测试效率。在线下压力测试时，需要注意 JIT 是否启用，-XX:CompileThreshold 的默认值是 10000 次，也就是超过 10000 次 JIT 才会启动，直接将 URL 编译成机器码，否则由于 JVM 是解释执行的，所以在压力测试时需要经过一定的时间进行预热，以便获取比较精确的 QPS 测算值。在线下压力测试时，可以选择对线上的访问日志（access_log）进行压力测试，这样可以保证正确的配比，因为不同的 URL 对资源的消耗不同，所以线下模拟的环境要和线上环境保持一致（一致的比例，一致的 URL）。

2. 线下脚本压力测试

线下脚本压力测试一般适合预发布机器，预发布机器一般没有用户的真实流量，用线下的 Jmeter 压力测试脚本进行压力测试同样要注意 JIT 是否启用的问题，以确保最真实的测试效果。

3. 线上引流测试

线上引流测试是主要的测试方式，它通过修改负载均衡器的权重因子，逐步将流量引入某台指定的线上机器。

9.3.2 两种常用的引流压力测试方法

1. 4 层负载均衡器权重比引流

线上引流测试单机吞吐能力，首先将负载均衡器的负载均衡算法修改成 round-robin weight，

接着在 lvs 所在的物理机上开发一个能够修改 lvs 配置脚本，然后公开一个（HTTP）接口，让线上压力测试系统进行调用，线上压力测试系统会根据经验值逐步发起 HTTP 接口的调用，将引流的权重逐步加大。线上压力测试系统提交给接口的参数包含权重和机器名，告诉 lvs 要多少权重、到哪台机器，可以手动停止也可以自动收集机器上的资源情况，如 CPU、CPU Load、内存、异常、GC 次数、RT 等。

2. 7 层负载均衡器代理地址变更引流

在大型网站架构中一般 4 层负载均衡器主要用于均衡负载，高效地转发数据包。7 层负载均衡器代理主要用于日志采集，线上压力测试系统会调用 Nginx 所在物理机上的能够修改代理转发的服务器地址（默认是本机）的接口（修改 nginx.conf 中的 proxy_pass 配置），线上引流压力测试系统会根据一定的时间周期，将应用集群中 Nginx 的配置地址逐步修改到目标机器上，如图 9-2 所示。修改完成后，要重新加载 Nginx 配置，直到流量足够为止（此时流量资源瓶颈开始出现，QPS 监控曲线从最高点逐步衰减）。对于 7 层负载均衡的引流测试方法而言，机器数量越多，引流的效果越好。引流可以从小流量开始逐步加大，直到峰值 QPS 出现。

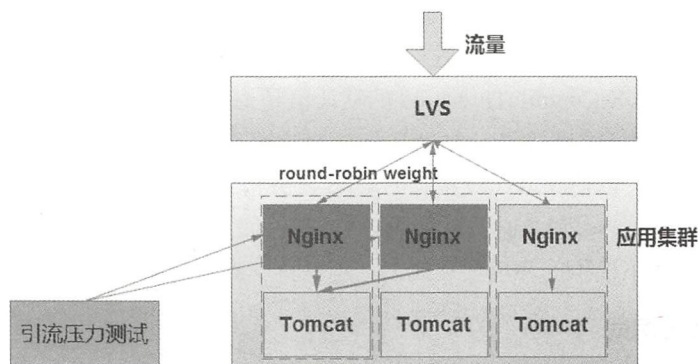


图 9-2

9.3.3 引流压力测试停止时间的判断

线上引流如果判断不准很容易出现问题，特别是监控系统汇总的粒度比较大，如监控间隔时间为一分钟，在一分钟内可能会出现很多问题，笔者在实践引流压力测试的过程中也遇到过问题。如图 9-3 所示，为了压力测试某个公共服务的吞吐能力，笔者将调用方（从 A 到 N）的流量逐步引至公共服务集群中的某台机器上，这台服务器的压力越来越大，超时开始出现，由于报警粒度比较粗（一分钟），加上压力测试期间应用方关注比较少，造成了一个故障。

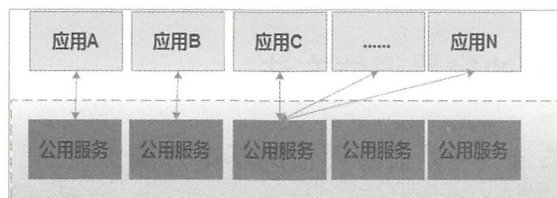


图 9-3

9.3.4 如何避免单机压力测试出现问题

从实际经验来看，需要从以下 4 个方面来避免单机压力测试出现问题。

1. 适当的流程保障

在单机引流压力测试期间需要专人“盯防”。针对某些服务的压力测试，需要多方一起保障。对于应用的压力测试，一般只需提请压力测试的人员在场，并且在压力测试结束的时候恢复现场（恢复到原来的流量比例），这些基本流程可以固定下来，作为一种制度来进行规范化的保障。

2. 时刻关注报警

在引流压力测试期间，单台机器上的流量会快速增加，当资源出现瓶颈时，超时（CPU、Load、远程、丢包）异常开始出现，这些异常的出现往往意味着线上开始出现问题。所以要时刻关注报警，必要时立即停止流量压力测试。

3. 登录到线上服务器关注异常

一般监控系统很难做到非常短的间隔时间的监控，另外监控存在一定的延迟，因此要确保没有问题，需要登录到引流的目标机器上观察日志和各种资源的表现。发现表现异常时，及时进行处理。

注：一般日志通过日志采集系统，经过实时的大数据处理，再到监控系统里，最终监控系统根据预设的报警规则进行报警。

4. 慎重对待公共服务压力测试

某些公共服务，例如会员服务，全网站都会依赖，当对会员服务进行压力测试单机容量时，大量应用对服务的流量会引流到某个单点的公共服务器上，导致下单关键链路上的应用出现大面积异常，进而影响用户下单。所以针对服务的压力测试，一定要将优先级调整到最高，并且需要关键应用的所有者进行线上观测，避免发生故障。

9.4 大型网站常用的容量评估方法

9.4.1 二八原则评估法——新业务评估的基本方法

其实在业务高速发展的时候，经常会遇到一个问题，如何在第一次做容量评估的时候，根据业务量就可以进行呢？第一次没有更好的答案，通常是根据二八原则进行评估。对于一个普通的营销场景普遍可以用二八原则评估法进行容量评估，当然秒杀、限时抢购、限时活动除外。对于秒杀、限时抢购活动，可以根据业务预计的时间进行调整。

$$\text{评估 QPS} = (\text{业务量} / 24 \times 3600) \times 80\% / 20\%$$

9.4.2 有历史数据参考的容量评估——GMV 线性比例评估法和 GMV 转化评估法

在做大促的过程中，往往会有两个关键的指标：总 GMV 和总 PV。通过这两个指标来得到各个应用、中间件和存储需要提供的容量。容量评估方法一般针对每个应用集群评估需要提供的容量。

1. GMV 线性比例评估法

GMV 线性比例评估法指的是，根据上一次大促活动时网站产生的 GMV 和将要进行的大促业务预计的 GMV 之间的比例，来扩充流量和 QPS。

注：如果是第一次做大促，那么可以根据平时网站产生的 GMV 和将要进行的大促的 GMV 进行评估。

如图 9-4 所示，监控系统给出的每个应用的峰值 QPS（Peak QPS），一般从分钟维度计算 Cookie Log，得到每秒处理完成的最高请求数。预留的 Buffer 指的是业务保障目标和技术保障目标的比例，一般技术保障目标会预留 30% 的 Buffer 给业务，也就是预留 Buffer=1.3，这样可以避免因为业务评估过小导致影响大促。



图 9-4

线性评估法简单、评估速度快，但是准确度极差。当网站流量变大时，会带来非常大的误差，不仅造成机器资源的浪费（钱能解决的问题不是大问题），而且造成大量人力资源的浪费

（需要做瓶颈的排查和定位工作），在较短时间内完成压力测试目标非常困难。线性评估的主要不足在于忽视了链路的差异，导致评估严重失衡。

在一次大促过程中，各种流量分布并不均匀，后台应用如订单管理、前台应用首页，以及活动页面的流量分布是不同的，前台应用之间的流量分布也是极其不平衡的，如活动页面和 Detail 页面的流量可能比搜索 List 页面的流量要高很多，但是如果按照相同的比例评估就会造成评估不平衡，甚至严重失衡，非常可能出现搜索 List 页面的容量评估过高，而 Detail 页面的容量评估过低，这种形式的评估可能导致大促失败。

2. GMV 转化评估法

为了弥补 GMV 线性比例评估法带来的评估严重失衡问题，GMV 转化评估法应运而生，它能够根据实际链路到订单的转化率来推测页面的 QPS。它还能够根据业务的表现推测出每个业务需要提供的 QPS。

可以将各个系统按照转化率的方法进行推理。

第1步，订单 QPS 预估。 $GMV/单价 = 订单数$ ，即一天内完成订单的请求数，再根据历史数据得出峰值订单 QPS。

第2步，根据转化率预估关键页面 QPS。

关键页面 $QPS = 该页面渠道到订单的 QPS / 到订单的转化率$ 。由于订单 QPS 由多个渠道聚合而成，对于 Detail 页面的评估，需要知道直接转化到订单的渠道的各自占比情况，再根据占比情况来确定由 Detail 页面产生的订单 QPS 所占的比例。因此 $QPS(Detail) = 订单 QPS \times Detail$ 产生的订单的占比 / Detail 到订单的转化率。

第3步，根据主页面和异步请求的比例关系，预估异步 Ajax 请求。

异步 Ajax 请求如果流到和主页面同样的应用上，可以根据应用监控获取主页面和异步 Ajax 请求的比例，进而预估异步 Ajax 请求 QPS。

第4步，根据主页面和非关键链路页面的比例关系，预估非关键链路页面 QPS。

同样非关键路径页面请求如果流到和主页面一样的应用上，可以根据应用监控获取主页面和非关键链路页面请求的比例，进而预估非关键链路页面 QPS。

第5步，预估爬虫 QPS。

爬虫也会有一定比例的访问，需要计算出大概的比例。要指出的是，爬虫的数量并不会随

着大促活动的变化而变化，所以爬虫 QPS 的预估一般不能简单地按照比例来进行，可以按照绝对值来进行。从实际观测数据来看，爬虫在大促期间占比会变小，因为其他流量变大了，而爬虫的爬取量一般相对比较稳定。爬虫日常占比较大，大促期间占比较小，是否计算爬虫 QPS，要看爬虫爬取高峰和大促峰值是否重合。如果重合，那么需要在峰值之内计算；如果错峰，那么爬虫爬取带来的流量可以忽略。

最终预估 QPS = 关键页面 QPS + 异步 Ajax 请求 QPS + 非关键链路页面 QPS + 爬虫 QPS

3. GMV 评估实战

做了多次大促备战之后，大促的量级越来越大，互联网要求的快速迭代慢慢和大促备战产生了一定的矛盾，评估误差越大，大促备战花费的时间越长，而每次大促花费的人力成本也越来越大，日常的很多快速试错、快速迭代越来越难以满足大促活动的需求。在容量评估的演进过程中，总发现实际的大小比评估的大小要小很多，这就是因为在容量评估的过程中存在比较大的误差，这个误差是去除 Buffer 之后的误差的体现。例如业务预估需要 1 亿的 QPS，为了防止合理的突发流量访问，技术保障目标往往提高 30%~40%，它会保障到 1.4 亿的 QPS。

通过仔细回顾和重新检查发现了以下问题。

1) 评估误区之一：单渠道全占式评估错误

从图 9-5 可以看出，有各种渠道可以产生订单，也是容量产生的源头和途径，图中三个页面 A、B 和 C 都有一定比例产生订单的请求，占比也不同，图中 B 页面是主要的下单渠道，下单的流量主要来源于 B 页面。所以从流量流向看，是 3 个渠道之和最终产生了订单的流量。

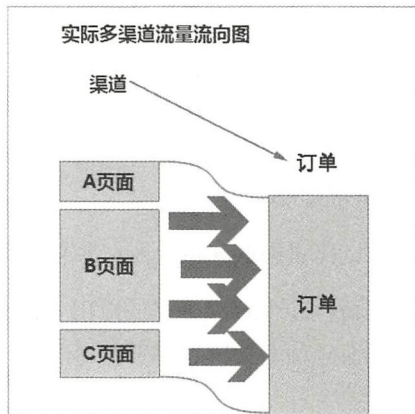


图 9-5

单渠道全占式评估指的是，当单个渠道放大成全渠道，错误地认为订单产生的流量完全是由此渠道产生的，从而放大了单渠道本身的流量，也就是该页面的 QPS 被错误地评估，如图 9-6 所示。

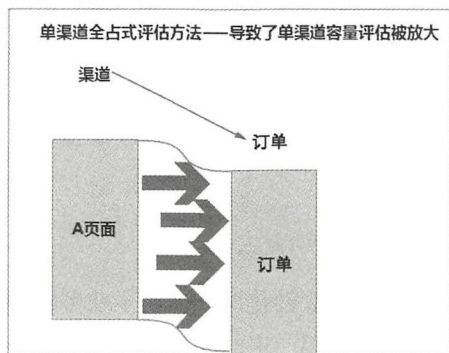


图 9-6

导致单渠道全占式评估的原因主要是，渠道比例监控缺失，开发人员无法获取比例关系，错误地把单渠道流量进行全占式计算，造成了评估结果变大。

如果 A 页面占比为 20%，B 页面占比为 50%，C 页面占比为 30%，那么 A 页面评估被放大了 5 倍，B 页面评估被放大了 2 倍，C 页面评估被放大了 3.3 倍，平均被放大了 3 倍以上。

要获取正确的评估数据，细分化的监控是非常关键的。

2) 评估误区之二：将 UV 转化率当作 PV 转化率

GMV 评估方式简化了链路的变化，根据评估点到 O 的转化情况，直接计算出评估页面需要的 QPS。在实际运算过程中，需要明确知道 UV 转化率的含义：在一定的时期内，评估页面的 UV 数量与最终下单的 UV 数量之比。例如，在 5 分钟内，List 页面有 1000 个用户访问，最终在 5 分钟内有 500 个用户下单，那么 L 到 O 的 UV 转化率是 50%；如果在 5 分钟内 List 页面有 1000 位用户访问了 10000 次，其中有 500 个用户产生了 1000 个订单，那么 List 到 O 的 PV 转化率是 10%。从这个含义可以看出，PV 转化率和 UV 转化率差别很大。在实际计算的过程中，笔者曾经将 Cookie 的转化率当作 PV 的转化率，PV 的转化率一般偏低，这会造成容量评估偏小。

3) 评估误区之三：忽略了页面的附属轻量级请求

一个页面包含数十个请求，静态资源的请求一般经过 CDN 静态缓存加速，流量会落到 CDN 节点上，页面除了静态资源请求，还有异步 Ajax 请求，这些请求很多时候需要落到和页面所属



的同一个应用上。在实际的容量评估过程中，很容易忽略这些看起来轻量级的请求，造成预估容量被缩小。

4) 评估误区之四：忽略了非关键页面请求

一个应用通常包含一些和转化相关的重要页面，同时也包含和转化无直接关系的页面，例如商品大图页面、商家介绍页面，这些页面实际上访问量也不小，也会消耗服务器的资源。通常在实际评估过程中，由于评估的是和转化相关的页面，很容易忽视这些非关键页面请求。

4. GMV 转化评估法的不足

1) 转化率度量采用窗口机制决定了误差大

业务监控采用 5 分钟窗口大小来计算转化率，是出于业务实时转化的稳定性来考虑的，用户从浏览页面到下单，实际上经过思考和一定的停留时间，在一个小的窗口内计算转化率随机性非常大，导致实时转化率曲线变化非常大，所以采用合适的窗口大小，转化率变化相对较小，在同比周期内相对稳定，避免因错误预警而增加人工成本。窗口越大，转化率测量和度量时间越长，因为 QPS 是每秒的吞吐量，经过窗口的统计值换算成秒级，误差通常会比较大。

2) 每个应用的特点不同，订单的峰值不等于流量的峰值

在一次大促活动过程中，每秒订单的峰值非常可能来源于活动和爆品，很多用户通常在促销之前，已经将商品加入了购物车，所以订单的峰值和活动相关的应用峰值是一致的。但是搜索类型应用的峰值通常来源于自然购物时间，所以将各个应用的峰值都按照订单峰值时刻的转化率来推理，结果通常是错误的。

3) 忽略了 PV 打点的丢失率

在 GMV 评估模型中，根据转化率来计算页面的 QPS，然后根据页面和其他请求的比例关系来获取整个应用的 QPS。转化率的计算分母是页面的 PV（或者 UV），分子是订单数，订单数的打点相对比较精确，由服务端直接计算，而 PV 打点根据页面发送的 Ajax 请求进行计算，存在一定的丢失率。具体的丢失率需要根据日常的比例获得，如果差异大，可以采用平均丢失率的比例进行计算。

4) 总体评估复杂度高

需要获取业务难以给予的数据，特别是远离订单的页面，评估是十分困难的，要获取 List 与订单的转化率是非常复杂的，把这个评估方法交给应用的所有者，几乎不可能获取到 List 的真实转化率，最终的结果可能是开发人员随便给一个转化率，导致评估严重失真。



9.4.3 流量占比评估法

转化率评估法需要获取各种转化率，而且最重要的是根本无法获取各个流量渠道到订单页面的真实转化率，评估复杂，可操作性较差。流量占比评估法简化了 GMV 转化过程中的各种计算和各种数据的占比，根据全站页面的 PV 占比进行评估，可以通过 GMV 的数值估计出全站的 PV，根据各个应用重点页面的流量占比，估算出重点页面的 QPS，再根据页面和异步 Ajax 请求的占比、页面和非重点页面的占比、页面请求和爬虫请求的占比，最终估算出应用需要的 QPS。详细评估过程如下。

第1步，根据 GMV 推导 PV。

一般业务方也会给出一次大促的估算 PV，但是为了安全起见，可以根据历次大促 PV 和 GMV 的比例关系，估算出将要进行的大促的 PV，并且和业务方给出的 PV 进行对比，取较大值，避免因评估过小导致系统出现问题。

第2步，根据业务 PV 打点统计各个页面 PV 的占比。

PV 一般通过页面上的异步 Ajax 请求获取，并且采集到大数据处理中心。可以按照各个应用的重点页面进行整理，最终获取各个页面的大促 PV，预估全天的 PV。

第3步，根据历次大促应用全天的请求数和该应用的 PV 的比例，计算出 Query/PV 的值。

第4步，应用上一次大促的峰值 QPS 和平均 QPS 的比例，画出两者的比例关系，如图 9-7 所示。

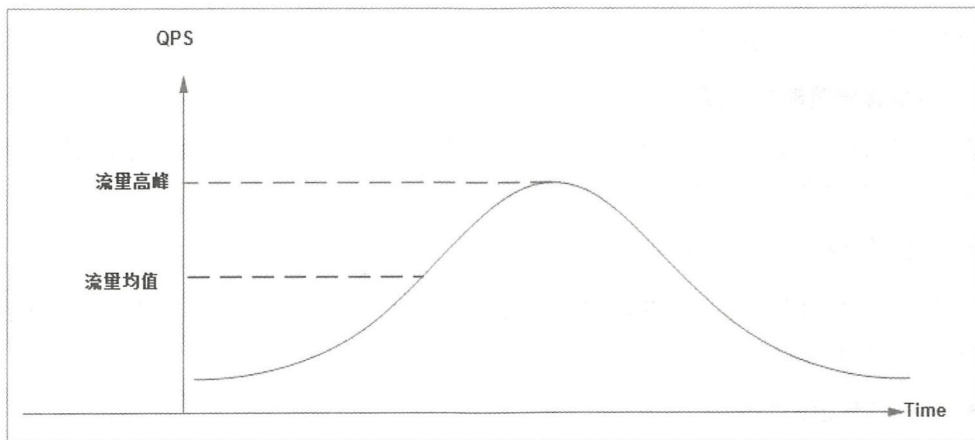


图 9-7



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

$$\text{QPS} = (\text{往年大促应用峰值 QPS} / \text{历次大促应用平均 QPS}) \times (\text{往年大促应用全天请求数} / \text{往年大促应用页面 PV}) \times \text{今年全天页面平均 QPS}$$

$$\text{今年全天页面平均 QPS} = \text{今年全天预估 PV} / (24 \times 3600)$$

1. 流量占比评估法的优点

评估方式比较简单，直接简化了大量的过程数据。流量占比评估法相比 GMV 转化评估法，屏蔽了转化窗口大小的统计误差，屏蔽了各种转化率，大大简化了评估过程。

2. 流量占比评估法的不足

根据 GMV 推导 PV，存在较大的变数。网站流量的来源类型比较多，例如外部投放、联盟投放、搜索引擎收录、付费投放等，各个流量的精准度不同。例如一次大促，付费投放了和用户需求匹配度较大的流量，这样相应的转化率会比较高，如果另一次大促，大量投放了转化率较低的联盟流量，那么会导致两次大促 GMV 和 PV 的比例差异非常大。

9.5 总结

1. 容量评估是一项系统工程

所谓工程，要求容量评估要有科学性和合理性，要有逻辑推理的过程，要有数据沉淀，要提供数据的评估依据，要根据每次评估的结果进行重新检查，以便做科学的修正。容量评估依赖于业务监控、系统监控和应用监控，这些监控系统的构建在整个性能优化体系中非常关键，所以容量评估是一项科学的系统工程，要引起足够的重视。

2. 峰值保障策略的必要性

在评估过程中，提出了一个新的问题，目前容量评估是为了满足峰值的流量而做的，而峰值流量的评估是否真的有必要，确实值得思考。

- 峰值的持续时间是容量评估的重要因素

这涉及业务的体量问题，如果峰值产生的流量持续一分钟，涉及订单数不多，那么容量评估可以考虑非峰值保障策略。

- 常态的峰值需要保障

常态的峰值指的是峰值持续时间较长，例如 30 分钟以上，这要利用体量来做具体衡量，流量较为平稳。



- 容量评估保障需要考虑口碑

互联网应用的一个重要特点是口碑很重要。一个大型网站，如果被用户认为技术比较落后，会影响长远的口碑。特别是大促开始的几分钟，爆品比较多，用户兴致很高地蜂拥而至，如果网站访问速度极慢，会引起不良口碑的恶性相传。所以在网站名气还不是很大的时候保障峰值时刻的访问速度，还是非常有必要的。

3. 容量评估要合理，而不是做到 100%精确

容量评估的合理性很重要，所谓大型网站的可扩展性水平也是相对而言的，随着吞吐量的要求越来越高，会出现各种瓶颈。从网络到存储，到带宽、电力，一个机房能够提供的吞吐量总是有限的。合理的容量评估，不仅可以大幅减少机器的数量，同时也能够减少人力的投入。容量要求越高，投入的机器和人力也会越多，容量评估是为了保障合理的峰值，而不是为了炫耀技术。依靠技术提高容量保障的目标，结果只能劳民伤财，通宵达旦地压力测试和保障是得不偿失的。容量评估的峰值保障策略很难做到 100%精确，它与人群分布、促销类型、范围、投放时间点和外部环境（政治环境、经济环境）都有关系，非常可能出现“今年双 11 的峰值比平均高 5 倍，明年高 3 倍”的情况，有一定的抖动是合理的。

4. 合理的容量评估离不开细分监控数据的支持

实践证明用没有分流占比的监控数据很难做到正确地评估，而单渠道全占式的容量评估又存在问题。无论是 GMV 转化评估法还是流量占比评估法都存在问题，峰值保障策略很难有规律可循。一次评估结果接近实际，并不能说明问题，因为峰值的多少带有随机性。



10

第 10 章

高性能系统架构模式

在大型网站架构过程中，存在一些共性的解决方案，这些解决方案在很多场景中得到了体现。例如，一个高性能网站一般都会用到缓存架构，缓存意味着高性能，通过空间换时间。近代 CPU 能力的高速提升，除了与核心架构有关，缓存的作用也功不可没。本章主要从宏观角度来看性能优化，好的架构是高性能的基础，只有从架构上解决问题，才能将高性能有效地持续下去。

从架构的角度来说，更重要的是体现高水平伸缩能力。单机吞吐量高并不代表能够通过增加机器数来满足高容量的需求，影响水平扩展能力的因素很多，所以有必要介绍一下高水平扩展能力的常见架构和部署模式。高并发的架构有些共同的特性，笔者将根据自己的经历进行比较全面的描述。高伸缩架构在实战中有很多难点，例如，库存系统的设计是目前高性能网站构建中最大的挑战，本书将点到为止，但是作为系统性架构的描述，本书要有所阐述，否则不成体系。从根本上来说，好的高扩展性架构，比局部优化带来的效果要大得多，如同人们经常说的：格局决定高度。



10.1 无状态架构

在高扩展架构的过渡阶段，笔者遇到的第一个问题是用户登录的问题。小型网站通常将用户的登录信息存在 Session 中，Session 是典型的有状态架构，而负载均衡会将用户流量均分在集群的每个负载中，用户的第一次请求分配到集群中的 A 机器上，后续的请求可能会分流到 B 机器上，那么 Session 的信息就会丢失。为了处理负载的随机分配问题，可采用 Session 复制的机制，Session 复制易造成水平扩展能力急剧下降，同时也会因为复制的延迟导致系统出现稳定性问题。

10.1.1 解决方案一——Session 复制

Session 复制发生在 Session 写入时，通过同步或者异步的方式进行 Session 复制，如图 10-1 所示。

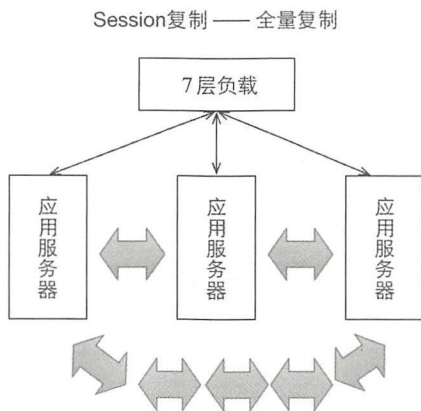


图 10-1

全量 Session 复制的架构有明显的伸缩性问题，随着集群的扩大，如果同步全量复制不仅会导致集群的吞吐能力受到极大的限制，同时也会造成客户端响应的延迟急剧增加。如果异步复制，会导致在 Session 复制没有完成之前，请求被分配到另外一个实例上，从而出现可用性问题，在大规模集群的情况下也会造成网络风暴，导致整个机房的容量和高可用性出现问题。

为了解决全量复制带来的复制成本和故障转移问题，WebLogic 应用服务器采用主从复制，当一个实例不可用时，请求会被分配到从实例上。



10.1.2 解决方案二——Session Sticky

负载均衡技术新增了 Session Sticky 机制，确保用户的访问通过负载均衡设备固定分配到用户第一次登录的应用服务器上，来解决 Session 复制的问题。但是这个方案最大的问题是故障转移能力缺失，当 Sticky 的机器不可用时，就会造成 Session 丢失。为了解决这个问题，类似于 WebLogic 的解决方案，采用主从复制，即当一台机器不可用时，用另外一台机器进行托管。

10.1.3 解决方案三——Session 集中式存储

无论是 Session 复制还是 Session Sticky 都带来了很大的伸缩性问题，Session 集中式存储是将 Session 信息缓冲到分布式集中存储中，应用服务器即使出现故障，也不会影响用户的登录，实现无缝的故障转移。同时 Session 集中式存储如 Tair、Redis，本身具有过期时间的自动管理功能，Session 过期的问题自然得到解决，如图 10-2 所示。

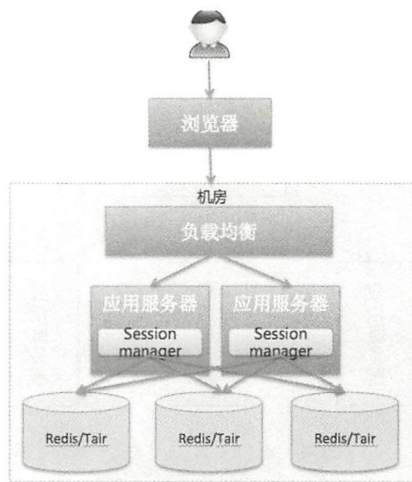


图 10-2

Session 集中式存储通常有两种实现方式。

第一种，通过 Web 容器插件（如 Tomcat、Jetty）来实现，Tomcat 有 tomcat-redis-session-manager，Jetty 有 jetty-session-redis。优点是对开发人员透明，开发人员还是跟以前一样使用 Session，不需要增加、修改任何代码；缺点是过于依赖容器，容器升级问题比较麻烦，好在容器升级是非常大的事情，不需要经常升级。

第二种，流行框架的会话管理工具，例如 spring-session 等，可以理解为替换了 Servlet 那一套会话管理，不依赖容器，不用改动代码。如果采用 spring-session，则使用 spring-data-redis 连



接池。Spring 使用范围较广，扩展性也好，缺点是只针对 Java 语言。

集中式存储需要考虑高可用的问题，一旦某个节点不可用，会出现严重的问题。如果是 Redis，可以采用主从复制架构，Redis 有强大的 Master 和多 Slave 复制的能力，如图 10-3 所示架构的缺点是，不能自动进行故障转移，需要手工进行切换。

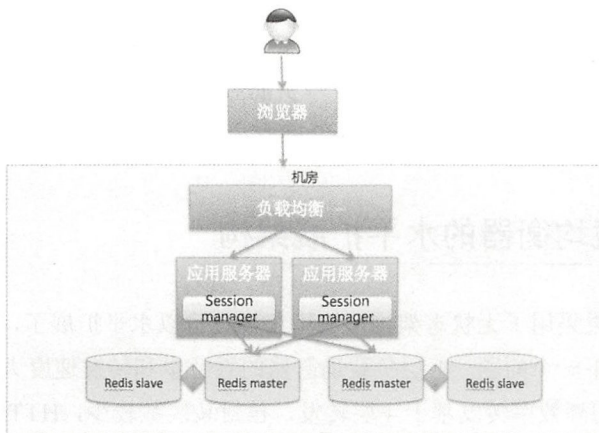


图 10-3

要实现自动故障转移，可以增加一层代理，在代理和 Redis 之间配置心跳检查，如图 10-4 所示。

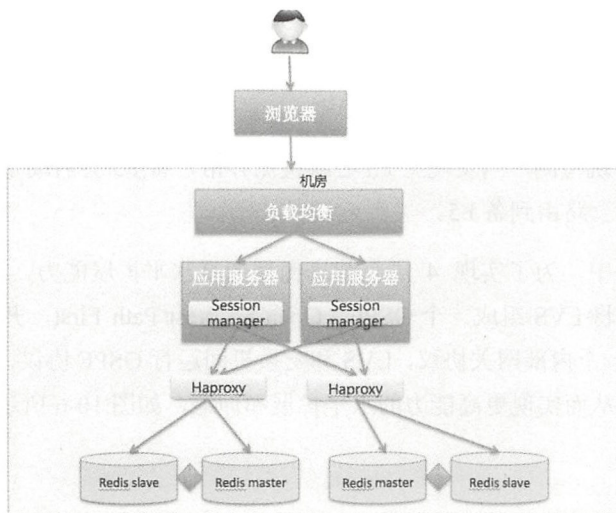


图 10-4



10.1.4 解决方案四——基于浏览器 Cookie 的无状态架构

Session 是有状态的，Session 复制的架构会造成扩展能力急剧下降，Cookie 的共享方式是，通过浏览器记录，将共享的具体信息存储在 Cookie 中，目前很多大型电商网站都采用这种架构。

Cookie 的最大问题是，大小存在限制，在有限制的情况下，维护 Cookie 比较烦琐，特别是在优化 Cookie 大小时，通常会下调已经没有维护的业务，但是修改 Cookie 是牵一发而动全身的事情，很容易造成大故障。另外要注意 Cookie 安全问题（要防止被篡改）和过期时间。

10.2 基于负载均衡器的水平扩展架构

一个应用集群如果采用了无状态架构，应用集群就可以水平扩展了，但是 7 层负载均衡器如何水平扩展就成了下一个问题。4 层负载均衡器的吞吐量和转发速度大大优于 7 层负载均衡器，4 层负载均衡器的高效率转发基于 4 层转发，包封装层数较少，HTTP 7 层封装转发效率较低，但是功能也较为强大，7 层转发可以做到基于 Cookie 的负载均衡。

7 层负载均衡器在大型网站中的作用通常有两个，一个作用是统计 QPS，将 HTTP 的所有访问记录在日志里面，QPS 可以基于日志进行，另外一个作用是通过 URL 的 rewrite 来实现静态化链接到动态化的重写。考虑效率问题，为了实现 7 层负载均衡器的水平扩展，通常在 7 层负载均衡器上架设 4 层负载均衡设备或者软负载服务器。

图 10-5 是 F5 的 4 层负载均衡的示意图，F5 基于主备（冷备）切换的方式实现高可用，备份 F5 和主 F5 进行心跳检测，当发现主 F5 心跳检测异常，备 F5 会启动全托管，主备之间的配置完全相同，路由器会路由到备 F5。

在很多大型网站中，为了实现 4 层负载均衡器的高水平扩展能力，通常通过 Anycast 和 ECMP 等价路由技术将 LVS 组成一个 OSPF（Open Shortest Path First，开放式最短路径优先）网络集群。OSPF 是一个内部网关协议，LVS 和交换机间运行 OSPF 协议，交换机上生成该 VIP 的等价路由 ECMP，从而实现更高能力的水平扩展和伸缩，如图 10-6 所示。

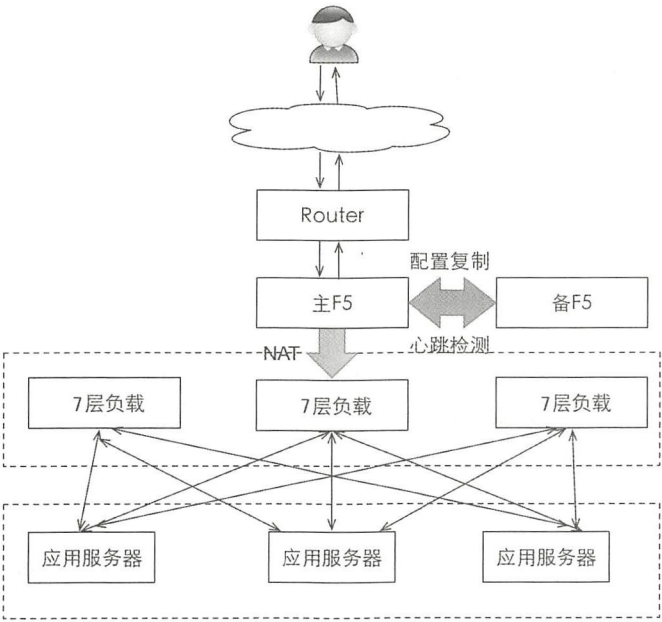


图 10-5

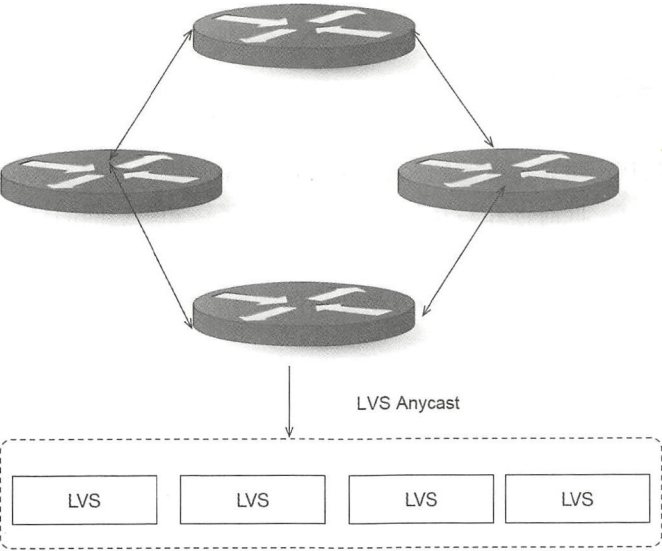


图 10-6

10.3 基于 DNS 的负载均衡

在某些场景下，为了降低部署的复杂度，同时在容量需求还不够大的时候，在技术储备还不是很好的情况下，可以采用 DNS 多 A 地址轮询来实现 4 层负载均衡的水平扩展，DNS 的负载均衡有如下的问题。

首先，故障转移相对而言较为困难，可维护性差。DNS 的负载均衡基于配置多 A 地址进行轮询，如果一台服务器失效，会导致将域名解析到该服务器的用户看到服务中断，即使用户按 Reload 按钮也无济于事。系统管理员也不能随时将一台服务器切出服务进行维护，例如进行操作系统和应用软件升级，需要修改 RR-DNS 服务器中的 IP 地址列表，把该服务器的 IP 地址从中划掉，然后等待一段时间，直到所有域名服务器将该域名到这台服务器的映射淘汰，所有映射到这台服务器的客户机不再使用该站点为止。

其次，负载倾斜可能会非常严重，由于 DNS 是从 Local DNS 到 Root DNS，一直到权威 DNS 进行递归解析的，每层都有缓存，一旦有缓存，DNS 解析请求不会发送给权威 DNS 进行解析，从而造成 RR 轮询不均衡。Local DNS 根据 TTL 进行缓存，而 Local DNS 存在很强的地域性，例如一个大型运营商在某个地区的 Local DNS 是相同的，由于运营的大小差异，会造成负载严重倾斜，服务器负载严重不均衡。

10.4 读写分离架构

读写分离架构是从小型系统过渡到大型系统的过程中常用的架构，其根本目的是减少写的压力，让读的性能更好，同时读和写互不影响，也消除了读和写之间的锁等待，减少延迟，从理论上能够将应用访问数据库的性能提升一倍。此架构适合在读一致性要求不高的场景中使用，Master 主库服务器负责包括读在内的事务型数据库操作，这是为了有强一致性的场景，Slave 辅库服务器负责无事务需求的读操作。

如图 10-7 所示，路由层的工作一般由中间件来完成，如淘宝的 TDDL 中间件就被用于完成此项工作。

在数据分布式存储——基于分库分表水平切分的架构中，也可以用读写分离的架构进行性能改善，短期内能够产生较为直观的效果。需要特别注意的是，在读写分离时要考虑程序本身是否做了事务控制，如果没有，会造成读写分离在不同的库中数据同步不及时，造成逻辑错

误。例如程序的逻辑是根据读的结果来判断是否可以写，当在备库读时，由于主库的写未及时同步到备库，因此读不到数据，判断为空，后续的写操作也无法继续，笔者曾经遇到过此类故障。

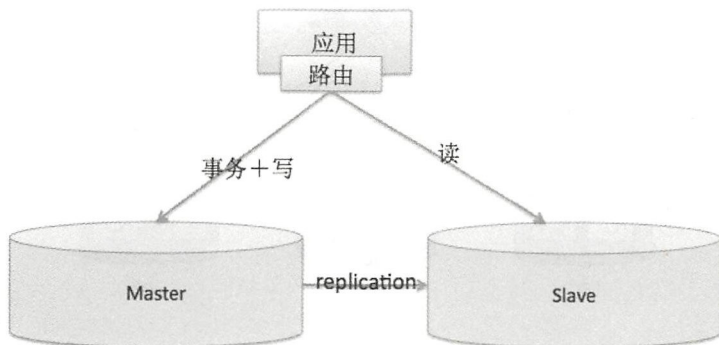


图 10-7

10.5 基于数据水平切分的水平扩展架构

数据水平切分主要针对写进行水平扩展，关系型数据库的写存在明显的上限，一般来说单库写数据时 QPS 不超过 3000，单表最好不要超过 4000 万行，否则查询、写的性能将急剧衰减。在强一致性的场景下，通常必须使用数据库进行数据的水平切分，将请求分散到不同的物理库中。对大型电子商务系统来说，有时商品数会达到数十亿，订单数也会达到数十亿，这种规模的数据，任何一个库都存放不下。将数据进行分库、分表存储，可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，从而提高查询速度。通常不建议将表进行垂直切分，垂直切分将字段进行分离，数据库的维护成本大大增加，同时对于开发使用极不方便，这就是像淘宝的 TDDL 等中间件，基本都是利用水平切分的方式进行数据可扩展的原因。具体分库分表方式如图 10-8 所示。

基于分库的水平分割：按记录行进行分割，不同的记录通过分库分表路由规则计算分别保存到不同的库中，将数据分离在不同的库中进行存储，每个子表的列数相同。

基于分表的水平分割：将表进行别名分割，通过路由规则进行计算，分到不同的表中，通常表名在数据库中将以别名的形式存在。

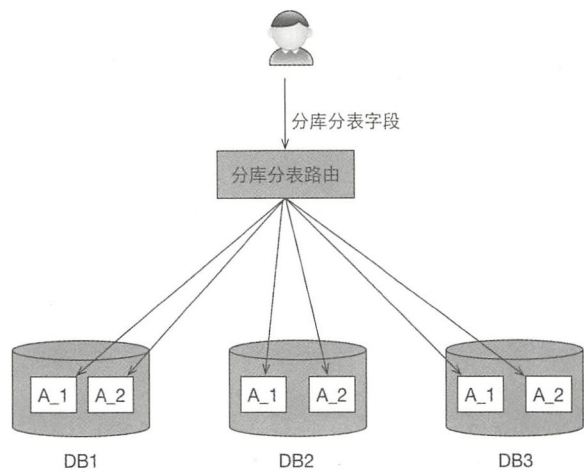


图 10-8

例如，将数据库分成 16 个库，每个库 16 张表，总共 256 张表。将商品进行分库分表存储，路由中间件主要完成 SQL 的透明解析、连接池的管理、并发控制和主备切换，对调用端透明，路由规则通常需要由使用方进行定义，可以根据自身的业务进行调整。例如，可以将商品 ID 的后 4 位作为分库位，中间 4 位作为分表位，也可以简单地将整个 ID 作为分库分表位。路由规则计算如图 10-9 所示，写和读都采用同样的路由规则进行计算。

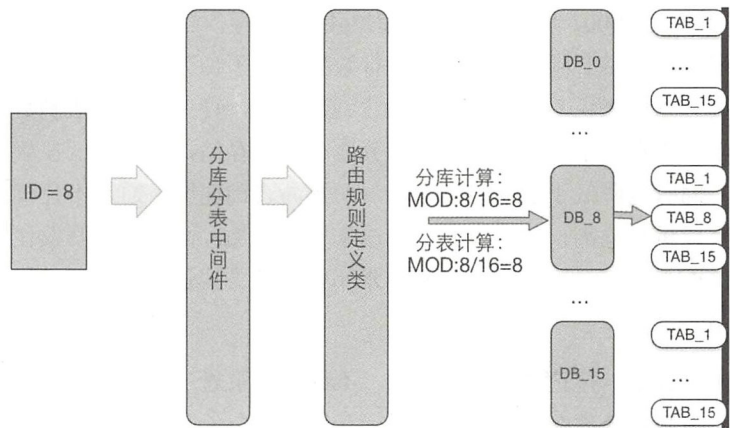


图 10-9

在分库分表的情况下，会产生跨库查询性能等问题，在这种情况下通常使用搜索引擎构造一个宽表，搜索引擎利用索引倒排的技术，基于行进行负载均衡，请求平均分发到某一行，行

中的每个列服务存储不同的索引，一行组成的服务器是索引的全量，最后由 merge Server 进行合并，如图 10-10 所示。

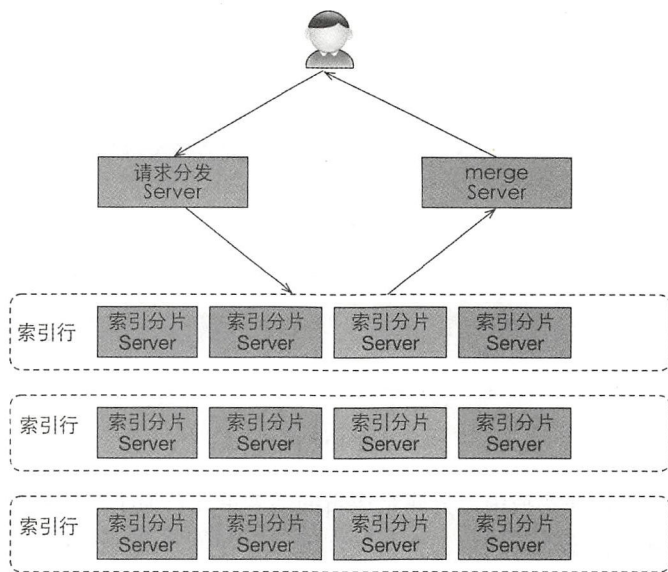


图 10-10

对于跨库事务，要做到强一致性必须采用分布式事务解决方案。很多时候，我们要尽量避免分布式事务，而采用本地事务解决问题。分布式事务解决方案，也可以通过补偿预案进行补充。

为了避免跨库访问，分库分表字段的设计要符合以下基本原则：根据业务场景真实访问的多少来进行分库分表字段维度的设计，以多的场景为最基本的分库分表维度。在实战中，通常会用很多维度进行分库分表的设计，例如买家、卖家都有查看订单的需求，通常以买家维度查看为主，同时买家用户体验的要求更高，订单表的维度设计需要以买家维度来进行设计，另外可以通过异步化消息的方式来进行卖家维度分库的设计。

另外，尽量合理地进行应用维度切分，应用（服务）分割访问可能也会带来跨库查询的问题（微服务架构同样存在类似的问题）。例如，在实战过程中理赔赔付的额度和保单的保额有一定的关系，理赔的额度最高不能超过保单的保额，我们需要从保单表中获取保额的信息。从架构的角度来说，可能会将理赔服务和保单服务进行切分，这就造成了分布式事务，需要全局事务进行保障。如图 10-11 所示，在分库分表字段上，为了避免跨库事务，将理赔单据和保单的分库数据放在一个逻辑库中，在设计时将理赔单据的后 4 位作为分库分表位。但是如果将理

赔和保单的服务分在不同的服务里面部署就会存在跨库事务的问题，理赔时会通过 RPC 接口调用保单的服务查询保额，此时本地事务无法跨应用生效，为了避免这种情况，可将理赔服务和保单服务进行合并。

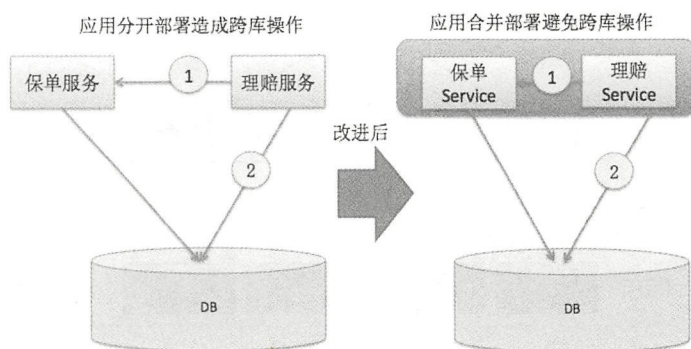


图 10-11

需要特别注意的是，在实战中，分库是逻辑的部署单元，为了节约物理资源，通常会将多个库部署在同一台物理机器上，往往都会交叉部署，笔者曾经碰到评价库和商品库放在一个物理机器上，由于评价和批量查询计算任务而导致相互影响的情况发生，在部署时通常要遵守以下基本原则。

- 尽量将重要的业务单独进行物理部署

重要的业务单独部署可以避免相互影响，因为一处问题会影响全局。

- 尽量将重要的业务和批量化的业务分开部署

单一的写和读性能对数据库的影响较小，特别是在索引设置不合理的情况下，会导致整体磁盘性能急剧下降，通常批量读取还会涉及幻读等问题。

- 尽量将非重要业务部署在一起

非重要业务放在一起，即使出现问题，也不会造成很大的影响。

10.6 缓存架构

缓存无疑是为了提升性能而生的，计算机发展到现在，计算速度越来越快，缓存功不可没。缓存架构是典型的以空间换时间的优化方法。

10.6.1 缓存的基本属性

- 命中率：表示缓存命中的次数占总请求数的百分比，这是缓存设计的重要质量指标之一。
- 容量：即缓存介质的容量最大值。
- 成本：即开发成本、部署成本、软硬件成本。
- 过期时间：缓存一般都有过期时间属性，也可以设置成永不过期。

10.6.2 缓存的分类

1. 按照存储位置分类

- 本地缓存

将数据缓存在本地，常见的本地缓存开源框架有 EhCache、OSCache、iBatis 和 Hibernate，本地缓存数据已经和这些缓存框架做了很好的集成，通过配置化的方式得到无侵入式的使用。

- 分布式缓存

常见的分布式缓存框架包括业界比较流行的 Memcached, Tair 的 MDB 解决方案对 Memcached 进行了很多借鉴。

2. 按照存储介质分类

- 内存缓存：将数据缓存在内存里。
- 磁盘缓存：Tair 的 LDB 解决方案是将数据缓存在磁盘（SSD）上，磁盘缓存的主要优势在于数据不容易丢失，即使在掉电的情况下也不会丢失，数据安全性比内存缓存要高。

3. 按照缓存的对象分类

- 页面缓存

对于商品详情页面来说，Web 页面缓存的 key 可以是商品 ID，对象是整个商品 ID 对应的 HTML 页面。通常缓存整个 HTML 页面的场景很少，一般的 Web 页面都含有大量的动态因素，例如商品的价格、库存的数量，这些数据都需要强一致性。通常在这种场景下会使用片段页面缓存，ESI（Eage Server Include）是片段页面缓存的常见解决方案，也是最早出现的 CDN 缓存 HTML 的解决方案，它已经成为一个通用的标准，Varnish 及 Squid 页面缓存解决方案都支持 ESI 的语法和标准。页面缓存也可以通过 CDN 进行动态加速。

- 对象缓存

通常对象缓存是为了减少远程网络开销和网络延迟，或为了减少 CPU 的计算以提升性能，

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

常见的如 DB 数据对象缓存和远程服务对象缓存。

10.6.3 缓存使用常见的问题和误区

缓存的特性是通过将计算好的结果缓存在内存中，后续在内存中直接获取，节省网络开销或者 CPU 时间，缓存带来的最大问题是脏读和数据的不一致。由于缓存数据的不一致带来了维护问题。笔者曾亲历了缓存带来的问题。

(1) 修改本地缓存对象，导致被其他线程读取，缓存对象被污染，引起故障。

本地缓存是将数据缓存在本地，由于线程通过引用修改了对象的值，其他线程读取时，读取了已经改变的值，导致故障，所以一定要确保本地缓存使用时，缓存的对象不会被修改，只是读取。

(2) 将缓存在同一时间点失效，导致缓存击穿，引发故障。

缓存完全被击穿是非常危险的，除非平时访问量很低，否则不要輕易地使缓存全量失效，或者在缓存设计时，要重点考量后端服务的承受能力，做好充分预估后再做这样的设计。

(3) 由于 Tair 缓存出现热点，导致 Tair 服务器崩溃而引发大规模故障。

笔者曾经遇到一个本地缓存击穿的案例，这个缓存是单 key 缓存，该 key 的访问量巨大。Tair 是根据 key 哈希到某一个 Tair 服务器上的，由于缓存单 key 热点对 Tair 某个服务器的访问量过大，造成 Tair 某台服务器崩溃，无法提供服务，进而引发大型故障。

缓存命中率是检验缓存效果的重要指标，命中率越高效果越好。命中率的高低通常和业务特点有关，本身不易变的业务数据命中率较高。缓存的命中率和其他因素也有关，例如过短的缓存时间，过少的缓存空间，导致被 LRU 交换出去，同时如果缓存对象设计不合理，对象极其易变，缓存频繁失效，最终也会导致命中率过低。

(4) 不考虑主动缓存失效，导致缓存对象已经被污染，引发故障。

在笔者经历的多项缓存架构升级的过程中，发现这个问题实际上比想象的严重。例如，在物流模板缓存架构升级过程中，缓存设置的过期时间是两天，这意味着两天时间内的物流模板非常可能是旧的数据。当时由于开始卖家设置的模板有问题，在几个小时之后修改了模板的内容，但是迟迟不生效，而在缓存架构设计的过程中，只是简单地认为卖家更新模板的概率很小，至少在一周以上才会进行模板的修改和更新，所以当时没有考虑主动失效方案，后来因为这个问题出了故障。经历这件事情之后，我们在缓存架构的规约中规定，所有的缓存架构方案设计

必须实现主动失效的功能。

(5) 缓存的对象过于复杂，因其中某些对象失效过于频繁，导致命中率急剧降低。

通常这种缓存对象包含了多个子对象，这些对象中只要存在一个易变的对象，就会被失效，导致缓存命中率过低。

(6) 缓存对象被过度拆分，调用端要多次读取才能完成业务的拼接，导致性能损耗。

通常这种情况发生的原因是缓存对象缺乏聚合度，聚合对象原本是业务需要的，然而需要多次 RPC 调用才能获取完整的业务对象。这种情况经常发生，网络开销始终是存在的。

(7) 内存型缓存，不考虑内存丢失兜底，可能因为掉电或者内存丢失，导致故障。

10.6.4 缓存使用场景

(1) 什么时候可以使用缓存？这是非常大的命题，不能用简单的一句话来表达清楚，因为有些业务对数据的一致性有很高的要求，所以笔者抛出以下几个观点，也是笔者认为的缓存使用原则。

- 在容量无法突破的情况下，考虑使用缓存来提升容量，否则尽量不要使用缓存。
- 对用户访问延迟有明显改善的情况，同时对优化 RT 有明显的效果，例如如果不缓存需要 2s，缓存之后只需要 100ms，可以考虑使用缓存。
- 使用缓存之前一定要了解缓存带来的问题，缓存不保障数据的一致性，要能接受这种不一致带来的各种问题，甚至资损问题。

(2) 使用内存型缓存（MDB）还是磁盘型缓存（LDB）

MDB 是内存型缓存的代表。

优点：

- QPS 极大，是 LDB 的数倍。
- 单份缓存，一致性相对较高。
- 响应更快。

缺点：

- 不保证数据的安全，在掉电的情况下数据容易丢失。
- 默认只有一份数据，没有备份，数据本身的可靠性稍差。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

所以使用 MDB 时要特别注意，一般需要 MDB+DB 做双防护，避免数据丢失导致数据无法获取。

LDB 是磁盘缓存的代表。

优点：

- QPS 比数据库大很多，比 MDB 小。
- 双份数据，数据安全性较高（不会因为掉电而丢失）。

缺点：

- 两份数据，一致性相对较低，当出现一份成功一份失败时，不会回滚。
- QPS 稍弱。

对性能有极高要求时通常使用 MDB，而对数据安全、数据可用性要求高时使用 LDB。例如，支付宝的 Session 信息就存放在 LDB 上，LDB 通常都会被当作数据源使用，不需要用 DB 进行兜底。

两者都需要考虑的问题：缓存场景下的一致性是无法得到根本保障的，要在强一致性的场景下，尽量弱依赖。例如，当数据库进行主备切换的时候，通过 Tair 来进行幂等的校验，防止因为主备延迟导致幂等失效。

10.6.5 缓存使用规范和原则

本地缓存需要确保缓存对象不会被更改，或者尽量少使用本地缓存，除非特殊场景，例如秒杀场景，对于 QPS 有极高的要求。

- 内存缓存一定要考虑双防护，后端一定要在 DB 层进行兜底。
- 内存缓存一定要加过期时间，一般以业务上数据变更的时长为准。
- 缓存的使用要实现主动失效方案，否则要特别说明。
- 所有为突破性能容量上限而设计的缓存架构方案，必须考虑兜底方案。
- 在设计缓存时，需要明确预估缓存命中率，过低的命中率导致的性能下降，比缓存数据不一致带来的风险更大。
- 避免 Tair 访问热点，Tair 的单服务器设计上限大约在 20 万 QPS，超过容量时 Tair 将提供服务。
- 设计缓存架构时，需要充分预估缓存不一致带来的风险，需要考虑快速发现方案、自动修复方案及兜底方案。



- 快速发现：监控上需要设计如何快速发现 Tair 的标记是否丢失。
- 自动修复：在出单时，校验数据库和 Tair 的标记是否一致，如果不一致，自动补充标记或者去除标记（如卖家退出运费险赠送时），将影响范围减小到最小。
- 超预估兜底：例如，业务上要求买家在前台下单页面看到运费险的标记时，一定要出单，所以在出单时，可以设置为如果超过天数不一致或者业务资损超过一定的限额，则禁止出单。

例如，对于运费险来说，卖家如果参加了运费险赠送活动，买家在购买该卖家的商品时就会出现运费险的标记，这个标记就是放在 Tair 上的，之所以放在 Tair 上，是因为考虑到运费险访问量巨大，用 DB 的方式支撑这么大的容量，需要巨大的投入。在这种情况下，无疑使用 Tair 是合理的，但是需要在设计时进行以下三个方面的设计。

- 设计缓存架构时，需要充分考虑缓存的分层影响，对于一致性要求高的元素和一致性要求不高的元素进行分离设计，充分评估影响面。例如工厂设计产品时，产品的价格是一致性要求比较高的，需要考虑分离设计，将价格进行分离，或者采用乐观锁版本号，如果版本发生变化，则从数据库中获取，如果没有发生变化，则从 Tair 中获取。
- 从 Tair 中操作数据时，一定要解析 `resultcode`，不要根据异常进行成功与否的判断。
- 客户端 Tair 不会抛出任何异常，即使在连接 Tair 服务器出现问题时，只会在 `resultcode` 中告知是连接超时还是错误。

10.7 近端架构

近端架构和缓存架构有点类似，近端架构主要解决的是由于 RPC 远程调用带来的网络开销，特别是在收敛比较小的机柜和服务器上，网络的瓶颈更容易出现。另外还要考虑线程池的开销，序列化、反序列化的开销，如果在应用层网络传输没有采用类似 `epoll` 的架构，还有内核区到用户区内存拷贝的开销。近端架构通常在性能要求极高的情况下采用，近端部署的逻辑通常在本地执行，逻辑相对比较简单，只有一些本地执行的代码。在实战中，如双 11 的宝贝详情页面对 QPS 有非常高的要求，对于营销优惠、运费的计算，还有运费险的推荐，可以对远程 RPC 调用的代码防盗宝贝详情应用进行部署，从而大幅度提高性能。

近端架构通常作为大促的常规预案，在日常正常流量的情况下，建议不要启用，近端部署的方式会带来维护性问题，也违反了架构的高内聚、低耦合的原则。



10.8 异步化架构

异步化架构是高扩展能力的常见模式，一般同步调用的架构会占用连接资源，特别是在 QPS 要求极高的情况下，连接资源占用会引起整个集群的连锁反应，异步化架构通常用在实时性要求不高的场景。在实战过程中，很多地方可以异步化，比如运费险出单的场景。大家都知道双 11 从零点开始，下单会出现极高的峰值，如果要在下单的同时出运费险保单，保险核保、出单动作是非常耗时的，保险的核心系统要达到和主交易同样的吞吐能力，是非常难的事情。通过业务分析可知，通常在用户发货之后才会有退货的场景，所以实际上从业务的角度来说，没有必要在下单的时候就产生保单，完全可以在主下单交易完成后再完成出单的动作。

通常，异步化架构通过事务型消息异步架构来完成：主链路业务处理完成后发布消息到消息中间件，消息中间件推送给消费者端应用，消费者端将异步处理任务给到数据库，再由业务系统扫描取出任务进行相应的业务处理，由于扫描任务表通常通过多线程和线程池处理，线程池的并发数可以调整，从而达到削峰的目的。具体过程如图 10-12 所示。

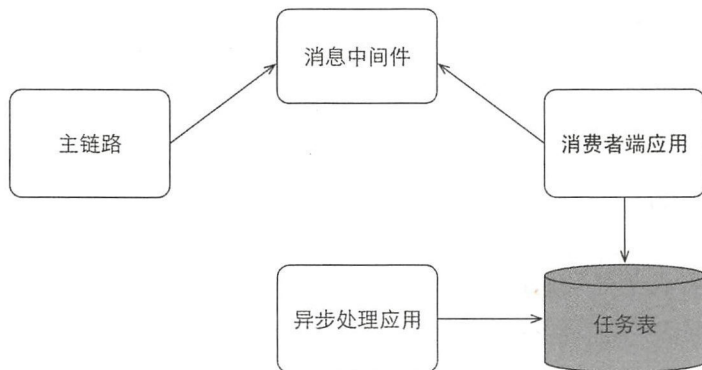


图 10-12

异步化架构的优点是，不仅能够削峰，而且能够通过异步化从强依赖变成弱依赖，从稳定性的角度看，减少直接依赖可以提高稳定性。

异步化架构的不足是，带来了维护成本，异步化往往需要后续的任务处理，需要建立任务表和任务扫描，维护的范围有所扩大，问题排查成本也比较高。



10.9 排队缓冲架构

在极高并发的场景下，如下单场景，如果直接将请求发到数据库，会造成数据库的崩溃，通常的做法是将请求放到队列中，进行排队处理，避免大规模冲击到数据库。既然是大型网站就免不了高并发的读写操作，很典型的一个例子就是秒杀，这种高并发的写操作，如果一下子都涌入数据库中，会导致数据库的压力非常大，从而导致客户端的访问延迟增加，即使不挂也容易造成数据库的死锁，遇到这种一拥而入的情况，就必须进行线性化操作。在代码层面上可以用锁机制来串行化，在分布式中可以用“消息队列”来串行化，而且还可以通过逻辑操作对消息队列进行动态的防洪和控洪。

排队缓冲架构，通常需要计算合理的阈值，超过阈值的部分使用队列进行排队缓冲，即将请求放到缓冲队列中进行串行化处理。一旦进入缓冲队列，用户端的响应时间会变长，注意要设置合理的队列长度，否则会造成用户体验变差。像秒杀这种场景，用户响应时间超过 3s，页面可能处于白屏状态或者旋转等待状态。如果每次响应的时间超过 10ms，那么队列的长度建议设置成 300 个，这样用户等待的最长时间不会超过 3s，具体过程如图 10-13 所示。

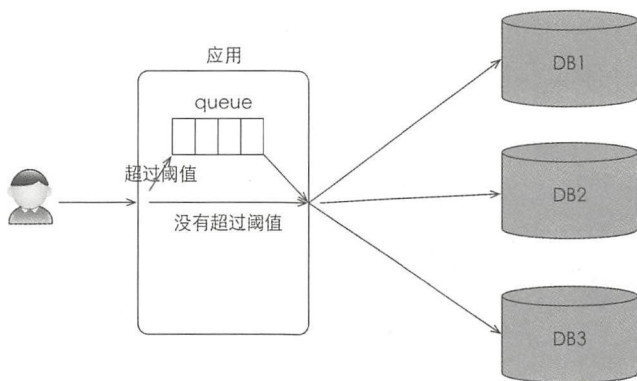


图 10-13

淘宝开源框架 Tengine 也采用了类似的机制，Tengine 可以设置限流阈值，超过阈值的部分不是直接被丢弃，而是放在一个队列里面，而且笔者在实战过程中发现了一个问题，当 QPS 压力测试到一定程度时，会一直处于一个稳定的状态，但是 RTT 会逐渐变长，而且没有任何错误的响应。通过了解原理，原来 Tengine 有个队列，当队列越来越长时，响应时间也会越来越长。



10.10 多机房架构

多机房架构用于提升机房级别的水平扩展能力，提高吞吐能力。在大型网站的构建过程中，单机房在电力供应、服务器存放空间、设备的连接数、机房整体的吞吐能力等方面都有明显的容量上限，还有可用性方面的问题。在笔者遇到的很多故障中，包括数据库被误删、机器坏死、交换机故障、其他网络故障、中间件故障及安全攻击等，多机房架构起到了十分关键的作用。

10.10.1 同城架构

同城机房的主要作用是通过双冗余能力的建设来提升可用性、提升灾备的能力，同时带来容量的提升。同城机房有热备和冷备之分，冷备只启用一个机房提供服务，当机房出现问题时，再启用另外一个机房。热备同时启用两个机房对外提供服务，淘宝早期也采用冷备的方式，将整个机房进行备份，缺点是成本太高，对于能够承担多少流量，需要不断演练，优点是架构相对简单，一个机房即为一个完整的单元，目前主流的同城部署架构都是热备，如图 10-14 所示。

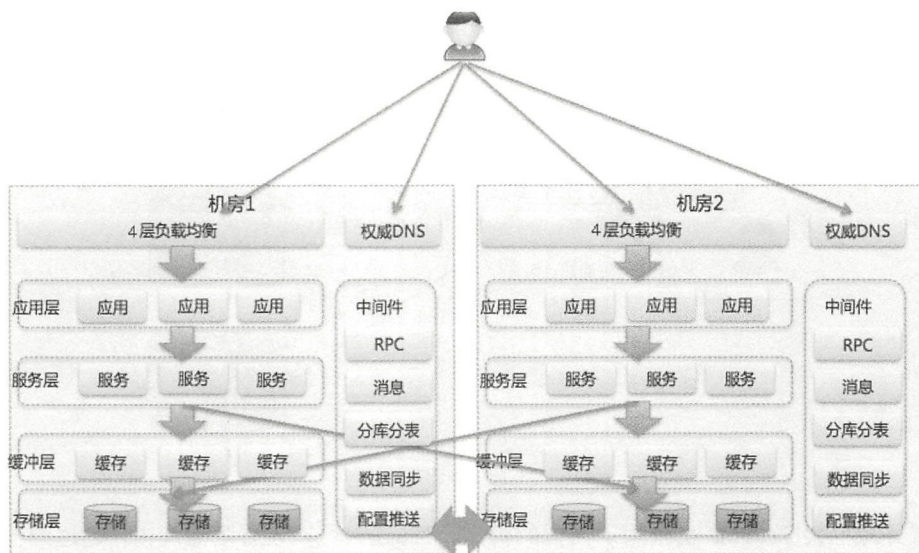


图 10-14

同城机房主要通过 DNS 的 A 地址轮询配置来实现机房级别的负载均衡，当某个机房不可用时，将 DNS 的 A 地址进行切换，对故障进行恢复。同城机房的难点不在于应用的故障转移，目前大型互联网的架构都可以做到应用是无状态的，所以在故障转移时，应用可以通过域名的



A 地址进行无缝切换，切换时间取决于 DNS 的 TTL。

同城架构的设计包含以下关键技术。

1. 数据强一致性保障

在同城架构中，机房间的距离较小，不一定是同一个城市。例如也可以认为杭州和上海属于同城机房，上海和杭州的距离大约为 200km，理论的网络延迟大约为 1ms，如果深圳和杭州的距离是 1300km，理论的网络延迟大约为 5ms，5ms 只是一个 RTT 的传输，如果一个数据包较大，可能需要多个 RTT 来回，那么访问数据库时会额外多 5~20ms，这种性能是无法接受的。保证高性能是很重要的，这决定架构方案。

强一致性保障有两种方案。

方案一：数据库跨机房部署，互为主备

如前面所说，跨机房访问的延迟是可以接受的，为了减小架构的复杂度，确保数据的强一致性，在数据访问上多采用跨机房部署，机房间数据库互为主备，机房间通过专线打通。所以从数据层面来说，数据库访问的吞吐能力并未得到提升，但是存储的机器数量增加了一倍，理论上可以做到水平扩展。当一个机房的数据库出现问题时，将数据路由切换到备库，每个机房的备份，主数据都是完整的全量数据，容灾切换时，原有的连接会受到影响。跨机房部署架构如图 10-15 所示。

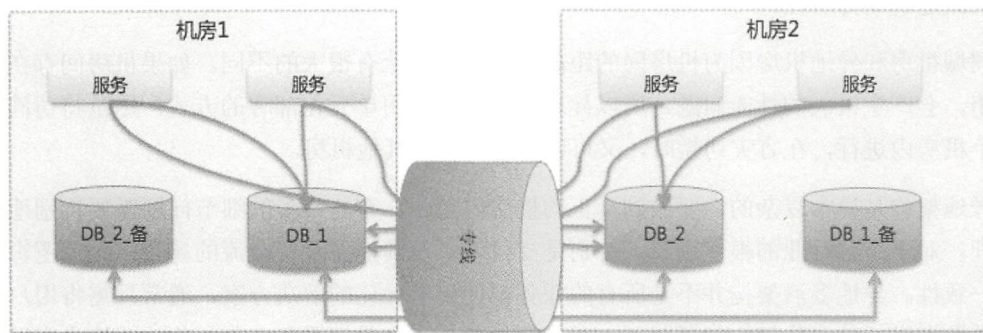


图 10-15

这里有一个细节问题，如果服务连接到出问题的数据库，其连接数已经接近连接池的上限，主备切换的作用并不是特别大，主备切换起作用的是新连接会连接到备库。还有一个方案是当发现主备切换效果不大时，可以重启服务，但要避免在访问高峰时切换，否则连接池资源可能很快会耗尽，形成雪崩效应。这也是大促期间很少启用机房级别容灾方案的原因之一，大促的



保障措施要完整地考虑到各种异常情形，做好充分的预案。

总的来说，数据库跨机房部署互为主备的架构，优点是架构方案简单，有一定的故障转移能力，不足是数据层的故障转移能力和灾备能力一般。这种架构是在机房间的网络延迟可以接受的情况下的部署架构，不需要特别的代码级别的兼容和复杂的一致性保障。

方案二：数据库冷备，主要考虑数据库的容灾

相对于整个机房冷备来说，数据库的冷备成本相对低很多，在正常情况下，可以确保数据的一致性。数据库冷备通过主机房的专线将数据同步到冷备的数据库，数据延迟要求较低，同城的机房确保都访问同一个数据库。此时应用、服务、缓存都自封闭，数据跨封闭访问。

2. 幂等失效处理

在主备切换期间有另外一个需要注意的问题，主备切换期间容易造成幂等失效，导致资损。一般的做法是尽量等 5 分钟左右进行切换，以确保数据同步的完整。

10.10.2 异地架构

异地架构是常见的高水平扩展能力的运维架构模式。同城机房将完整的业务数据全部放在一个机房里，包括交易、商品、导购、物流都在一个机房里，如果一个机房全挂，所有的业务都会受到影响。随着业务规模的扩大，一个城市的电力、机房的空間等无法满足需要，这是异地架构改造的动力所在。

同城机房和异地机房因为机房间的距离差异，架构上有很大的不同。如果机房间存在跨机房调用，会产生极大的性能问题，所以异地架构一般采用单元化部署的方式，尽量将访问切分到一个机房内进行，在容灾切换时，又可以无缝切换到其他机房。

异地架构是异常复杂的，要做到真正的机房内封闭，有非常多的细节问题需要特别地设计和处理。对于金融行业的很多场景，特别是支付场景及其他涉及资金流的场景，处理逻辑都需要强一致性。异地多活架构并不是所有的业务都使用单元化的解决方案，通常只需将用户直接相关的数据进行单元化架构部署即可，其他的如商品、会员等服务，单元化的改造成本很大，将买家和卖家的数据都在一个单元内封闭访问，会带来很多问题。例如，淘宝将买家交易进行单元化架构的改造，确保一个用户在商品搜索、查看、购物决策、交易的链路等环节能够全部走通。将所有与这个用户相关的数据都放在一个单元里，基于交易的改造主要问题还是交易的写压力很大，容量上有很大的瓶颈。同时从用户体验角度考虑，交易是非常关键的环节，用户访问性能越好，成交转化的可能性越高。



异地架构如图 10-16 所示。

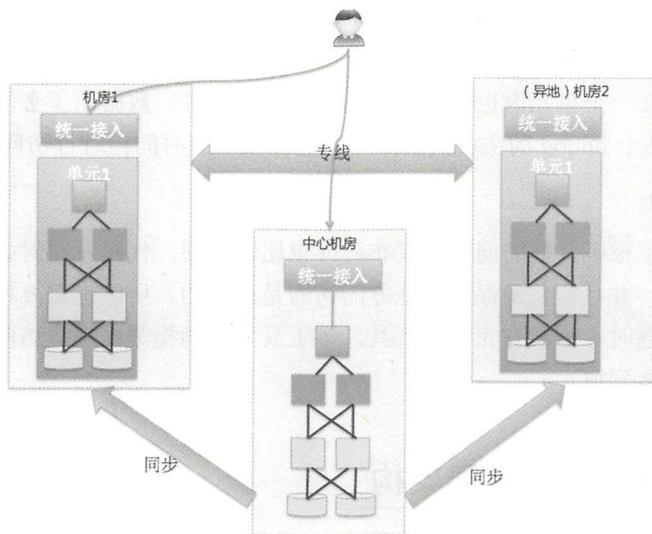


图 10-16

异地架构关键问题和关键技术如下。

1. 单元数据如何切分

单元数据切分一般从用户维度进行，如 Facebook 的多数据中心，是基于用户维度进行切分的。在电子商务平台上，购物车的数据以用户为维度，这个数据可以封闭在一个单元里进行访问，但是商品、卖家等信息是归属于卖家维度的数据，这些数据封闭在一个单元里进行访问或者直接跨机房访问，跨机房调用会带来很大的延迟问题，这在性能上是无法接受的。这些公共数据在一个单元内访问，需要同步来实现，写在中心机房完成，再通过专线同步到各个单元机房。

2. 用户访问路由

按照买家维度进行单元封闭访问时，要确保写和读都在这个单元内，否则如果出现写在两个机房，就会出现问题。所以路由时要确定用户访问的规则，这就是图 10-16 中统一接入时遵守的规则。这个规则要考虑多种情况，可选的规则包括按照用户的常用收货地址、用户注册地和用户的 IP 地址进行路由选择，通常会按照用户的收货地址进行确认，即使用户异地出差，路由规则必须和用户常用收货地址所在的城市保持一致，当然这样做可能会牺牲一点用户访问性能。



3. 延迟保障

按照买家维度切换和归属单元的方式，可以将用户产生的数据放在一个单元内封闭访问，但是卖家的数据、商品的数据，要做到在一个单元内访问，需要单元的机房和中心机房之间存在数据同步，确保在一个单元内也能获取商品、卖家的信息。数据同步必须确保很小的延迟，用传统的 MySQL 基于 binlog 的主备同步，不能确保在很短的时间内进行访问。

4. 发布和维护

异地架构带来了其他的问题，发布验证也比较麻烦。代码发布时必须确保所有的机房内的代码是一致的，并且需要确保多个地方的访问是正常的，所以也需要构造一个高效的发布验证系统。排查问题时，需要有完善的工具，该工具要明确检测出用户访问在哪个机房，数据在哪个机房没有同步到位。

10.11 基于服务的可扩展架构

基于服务（SOA）的可扩展架构更多地强调了复用性、组件化、高内聚、松耦合，解耦思想最终落实到架构设计中是业务能力组件化，组件间的交互通过服务接口进行，粗粒度的服务本身就体现了松耦合，解耦不仅包括组件的应用层，还包括数据库和数据层，都能够自成一套，可以独立进行需求、设计、开发、测试和运维的全生命周期管理。

基于服务的水平伸缩区别于传统的基于 soap XML 报文的伸缩架构。基于 RPC 的服务伸缩架构是点对点的，是一种去中心化的架构模式。传统的 SOA 架构需要基于 HTTP，一般 HTTP 需要经过 7 层代理服务器进行转发，从性能上来说 HTTP 的报文无论是基于 JSON 还是 XML，在性能上都有很大的优化空间。在大型互联网的分布式架构中，普遍采用类似 Dubbo、HSF 框架的架构，可以很好地支持更多报文压缩协议，例如 Google 的 ProtoBuffer 协议，基于二进制的 Hessian 协议。ProtoBuffer 协议通过索引表的方式，在交互时大幅压缩传输的内容，而 Hessian 协议则在压缩上取得了很大的优化。

在大型网站中采用基于服务的架构，目的如下。

1. 业务垂直化，高内聚低耦合，依赖弱耦合

在业务发展初期，人少、团队规模小、业务简单，业务上的诉求更多的是快速功能迭代、快速上线，在简单的业务背景下，架构的分解也是非常粗粒度的，也是合理的。但是当业务发展到数百人的时候，业务需求复杂度大幅增加，之前的一个功能从头到尾都是几个人在维护，变成很多功能每个团队都需要维护。典型的案例是订单费用计算和物流费用计算，随着业务的

发展，物流的费用在很多地方都需要展示，如搜索列表页面、商品详情页面、购物车下单页面、订单列表页面等。在早期的时候，可能这些展示需求都由一个人来维护问题不大，当分散到各个小团队的时候，每个人都必须对物流的逻辑有所了解，但是每个人理解的逻辑和获取的信息不同，可能造成用户看到的费用不一样，用户体验极差。所以需要功能内聚，由专门的团队来维护，提供服务化接口，供展示方调用。

2. 服务水平伸缩，和 Web 应用解除耦合

从伸缩性角度来说，如果功能耦合在一起，特别是 Web 系统和服务耦合在一起，带来的最大问题是伸缩困难。大家都知道动态 Web 页面的性能由于页面渲染带来大量的 CPU 消耗，导致性能很差，Web 页面的 QPS 远比单纯的服务的 CPU 计算量要大很多，杂糅在一起往往导致整体的性能很难提升。将服务和页面解耦，服务可以水平伸缩，在制定性能优化策略时也可以分而治之找到瓶颈所在。

3. 可维护性提升

从高效功能开发角度来说，基于服务的架构最大的好处是将功能之间的强耦合变成松耦合，可维护性更强。基于以前业务二方库的实现，将重要的业务逻辑放在二方库里，当二方库需要升级时，将所有的依赖全部梳理出来，如果有些二方库没有更新，就不能立即发现问题。笔者曾经经历过，某个系统依赖一个重要的二方库，一个月以后，该系统发布时发生了重大故障，而且当需要回滚时，不知道哪个二方库版本是对的。而基于服务的架构一旦发布，依赖的系统立刻有感知，也比较容易发现问题，便于及时修复。

4. 去中心化，点对点直连

传统的基于 SOA 的架构基于 soap、restful 接口，需要经过 7 层 HTTP 的转发进行负载均衡，水平伸缩时需要考虑 7 层负载均衡，部署复杂度相对较高，通过去中心化的 SOA 框架如 Dubbo 实现点对点直连，避免了 ESB 带来的伸缩性问题，水平伸缩能力更好。

5. 传输报文格式优化，性能高

不同于 XML 的报文格式，基于传统的 SOA 采用 XML 作为报文中介，传输内容多，接收方报文解析需要消耗不少 CPU 资源。XML 报文有首尾对称的节点名，造成空间浪费明显，相应解析的效率也会降低。而互联网常用的 Duboo 中间件，支持 Hessian 二进制压缩报文，报文更小，解析效率更高，还支持 Google 的 ProtoBuffer 协议，基于 index 的数据字典，通过双边的 index 对照来翻译出对应的方法名、节点名，从而大幅减少 CPU 消耗。

10.12 日结架构

日结架构是在金融类场景下常见的解决方案。在金融场景下一般需要依赖机构的能力，例如互联网保险平台，需要通过机构进行投保的核保和保单的生成，如果通过机构实时地交互，金融系统会承担很大的压力，无论是泛金融的保险场景，还是银行类的转账场景，系统的承压能力非常有限。日结的方式首先在业务上要得到认可，认可中间层先将业务进行处理，然后在 T 日日结给机构。特别是在投保的场景下，核保规则可以前置在中间层业务里，风险也由中间层来把控，并且风险实际是可控的。这种场景不适合于像车险、寿险这种复杂的核保规则，车险的核保需要与行业平台进行交互，业务平台无法直接和行业平台进行交互，并且有些数据业务中间平台很难获得，如车险的出险记录、交通违法情况、续保的情况等。

日结架构的使用场景总结如图 10-17 所示。

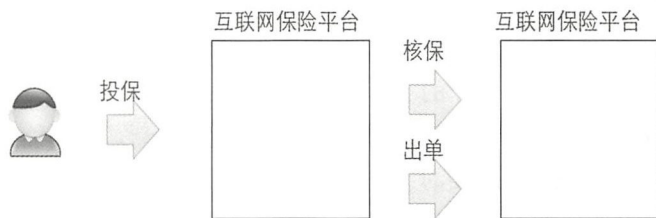


图 10-17

是否能使用日结架构，取决于平台的业务控制能力，如果核保规则可以前置处理，那么用日结架构问题不大，可以作为常用的模式。日结架构要求业务平台具有很强的掌控力，资金安全保障难度大，例如平台计算了错误的价格，如果跟机构交互，机构会对计算的订单金额进行校验，日结架构削弱了资金结算复核能力。

日结架构的优点：交互解耦、直接依赖变成弱依赖、松耦合，性能和稳定性都有很大的提升。不足：对业务场景要求高，需要业务有很强的掌控力，时效性较差，容易出现资金安全问题。

如果核保必须在机构端完成怎么处理？在实际保障过程中，也有方案来解决，可以将核保动作后置异步化，但这也要看业务的接受程度。

10.13 热点避免架构

热点是高性能架构常见的问题，所谓的热点通常指同一个数据被多次访问和更新，而且访问量巨大，超过了单点服务器的容量上限，由于访问集中在一个单点上，导致访问负载无法均衡。常见的分布式架构是指访问被均匀地分散到不同的机器上，每台服务器承担的访问容量大体比较均衡。在以下情况下，会明显出现热点效应。

1. 电商爆品的描述场景

尽管大多数架构会采用集中式缓存架构来进行高容量访问的承载，但是由于缓冲的 key 是商品 ID，一个商品缓冲会集中到一台服务器上，热点商品信息访问很可能超过单点缓冲服务器的容量上限。例如 Tair 的单点访问容量上限是 20 万 QPS，一个数据库分库的访问上限可能只有 3000~5000 QPS。在这种情况下，一个秒杀商品的库存信息更新，对应更新数据库的一条记录，而且在更新时为了防止并发，很可能要加行锁，无论是悲观锁还是乐观锁，如果访问量大会出现数据库访问超时。可以说电商的秒杀系统架构是高性能访问最难的问题，特别是像淘宝这种量级的网站，不仅要提供高性能的访问，还要提供有效防止库存超卖的措施，中间某个环节出问题，非常可能出现库存超卖。同时要考虑稳定性问题，库存对应的数据库如果要和其他业务部署，最好将库存数据库进行部署隔离，避免由于库存系统的问题，导致整个业务不可用。在实践过程中，要完全避免库存超卖很难做到，但是确实可以将库存超卖的概率变得极小，秒杀系统的具体设计可以参考阿里官网发布的双 11 系列丛书。

2. 新闻热点场景

近几年随着娱乐产业的急剧发展，某些明星热点新闻被微博等媒体曝光之后，由于互联网的病毒式传播能力，导致在一个瞬间，同一条新闻的访问量急剧上升，出现热点访问的问题。

3. 金融类的热点账户场景

在金融领域，热点账户的问题是非常常见的，在活动期间可能存在大量的用户同时向一个账户转账，这个账户对应数据库中的一条记录，需要频繁地对账户的余额进行更新，分库分表时大多数按用户维度进行切分，而热点账户在同一个数据库实例上，从而造成大量的锁竞争和锁等待，导致 QPS 过低。

热点问题的解决方案基本是通过空间来换时间的，无论是金融类的热点账户、微博类型的访问量巨大的大 V，还是热点新闻类型的业务，都是通过空间的分散来解决的。热点大部分都会出现在存储上，缓存架构和数据库架构尤其明显。

对于热点新闻类型的业务，笔者通常将这些新闻缓存到集中存储的服务器上，由于一条热点新闻通常会 Hash 到同一个服务器上，而一台服务器的处理能力是有限的，此时需要考虑如何对该热点进行分散处理。本地缓存是最常见的做法，也就是将热点的数据提前缓存到本地服务器，将数据从集中的缓存服务器上，比较均匀地分散到本地的数十台服务器或者数千台服务器上，从而有效地分散压力。

本地缓存架构要防止本地缓存全量失效的情况，笔者曾经经历过，本地缓存的 key 规则发生变更，导致全部的请求压到后端的热点存储上，如图 10-18 所示。

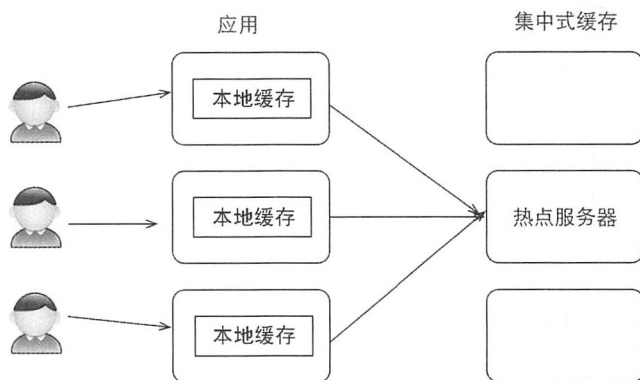


图 10-18

本地缓存架构的优点是，有效避免了远程缓存的单点服务器的热点访问压力，由于是基于内存的操作，存取速度快，有效地缓解了数据库的压力。缺点是本地缓存维护困难，很容易出现本地缓存穿透到远程集中式缓存或者数据库的问题，例如发布时忘记做灰度或者分批发布，导致本地缓存全部失效穿透。

本地缓存架构要考虑诸多的后续维护问题，尤其要注意以下几点。

- (1) 要防止本地缓存元素 key 的构造规则发生变化和升级，升级时要考虑灰度切流或者提前预热。
- (2) 避免由于本地缓存穿透导致热点集中崩溃，最终导致严重的故障。
- (3) 要防止本地缓存对象被修改，导致对象被污染。
- (4) 要特别注意灰度发布，不能一次性暴力重启集群中的全部机器，要考虑本地缓存的预热方案。

在金融场景下，热点账户的问题通常可以通过业务处理方式的变更来解决，例如前面提到的异步化架构，通过分析业务场景将实时性要求不高的场景进行异步化，再通过异步记录的扫描逐个消化，将同步带来的集中冲击改成控制固定并发的压力，这种情况通常在业务上想办法，通过业务缓存来避免大量瞬间的操作发生，具体方案如下。

方案一：日结批量入账

在 $T+1$ 日，将热点账号的操作经过平滑削峰处理进行批量平滑入账，类似于异步化架构，这种架构通过业务上的缓存方式来实现。该方案的优点是，账户热度低、系统压力小，缺点是实时性差。

方案二：缓存汇总入账

汇总入账是金融系统经常采用的业务分散的策略，通过时间换空间来减缓集中访问的压力，银行在 T 日的时间点完成所有成功交易汇总统计，通过交易凭证计算出总账，然后一次更新到结算账户，同时补充账务明细，用于明细对账。这实际上是异步化架构的一种具体业务形式的实现。

优点：数据库压力小，只要在 T 日汇总计算出结算金额，再与流水记录进行核对，就可以将汇总金额更新到指定账户。

缺点：结算金额不能实时更新，在 $T+1$ 日才能看到结算金额，结算账户的资金并未实时结算更新， T 日的交易款要到 $T+1$ 日才能结算到账。同时用户体验会比较差，用户发生了转账行为，但是当时总金额并没有发生变化。如果转账时账户是可以用的，告知用户转账成功，在实际转账时却发生失败，可能很多用户无法理解。

11

第 11 章

大促保障体系

11.1 大促保障概述

11.1.1 大促保障简介

大促是大型网站非常重要的营销形式，营销的作用是通过薄利多销的方式来吸引更多用户的关注，拉新是大促最重要的目的之一，运营的三个重要工作是拉新、促活和转化。像大促运营做预热工作，就是典型的先将用户圈定，在大促当天通过多种运营活动进行召回，如定点发优惠券、给用户发送营销短信等。大促当天转化率会有很大的提升，同时也意味着流量和并发会有所爆发，所以在大促环境下，对系统的容量要求很高。通常大促性能保障工作，跟业务或者运营的玩法直接相关，要做好保障工作，必须对运营的每个细节进行详细的了解，需要找到能配合运营的关键点、关键链路，将关键链路保障好。大促的保障工作是一个系统化的工程，包括项目管理、组织保障、风险保障、容量保障等具体的工作，如图 11-1 所示。

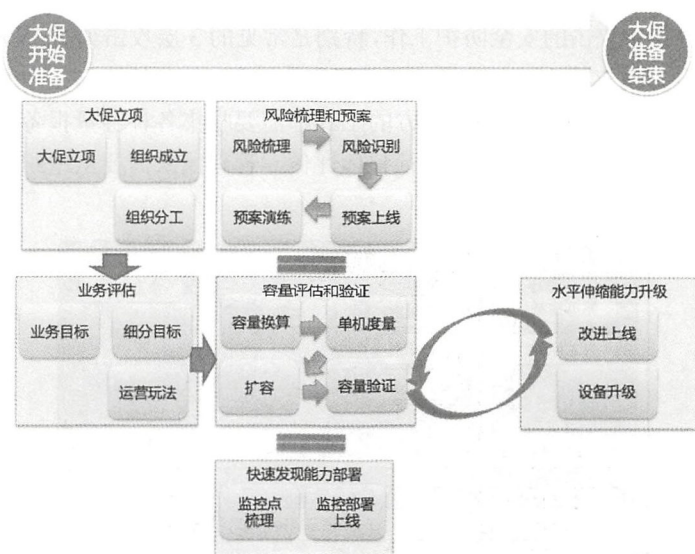


图 11-1

大促保障是性能优化的重要力量，大促对容量的峰值处理能力有很高要求。性能优化在很多时候要注重投入和产出的关系，高性能架构往往和高伸缩性架构有紧密的联系。本章将重点介绍大促的整体玩法和细节工作，辅以案例，加深读者的印象。同时由于大促保障本身是成体系的，而大促保障并不只是性能相关的课题，还有风险保障体系和资金安全保障体系，所以为了更成体系，本章也花了一点篇幅讲述风险保障和资金安全保障的内容。

11.1.2 大促保障整体流程

大促保障工作需要耗费不少的时间准备，要把它当作一个项目进行整体推进，一般都需要经过如图 11-2 所示的主要流程。实际的流程远比图中的复杂，需要很多人员参与。

从技术的角度看，首先整个项目需要有一个虚拟的团队负责，特别是涉及整个公司级别的大促保障工作，需要每个事业部出一个负责人，由负责人推进整体项目。负责人需要每周发送项目周报，包含整体封网时间、发布冻结时间、关键问题的收集、全链路压力测试计划等。架构师主要负责大促需求的关键架构设计。开发工程师主要完成项目需求，做好关键链路的依赖梳理、风险的识别、风险的修复及最终的风险预案的产出。负责性能的工程师必须配合全链路压力测试，准备好压力测试脚本，以及问题排查脚本，提前对大促关键链路做好细致而完备的监控，既要有总量的监控，也要有细节的监控，涉及资金安全，必须提前部署核对脚本、止血应急熔断开关。运维工程师需要协同开发工程师进行容量的评估和机器的采购。安全工程师主

要负责应用的安全、恶意攻击的安全防护工作，特别是常见的 3 层攻击如 SYN Flood 攻击、DDoS 攻击、7 层 CC 攻击、越权攻击、SQL 注入攻击和 XSS 攻击的防护工作。中间件工程师需要从工程师那里收集容量信息。公共服务提供方的工程师需要收集各种容量报备信息，并且做各种预案。数据工程师需要对数据库的容量进行整体评估，管理团队需要从组织上进行保障，制定制度，把握整体风险。

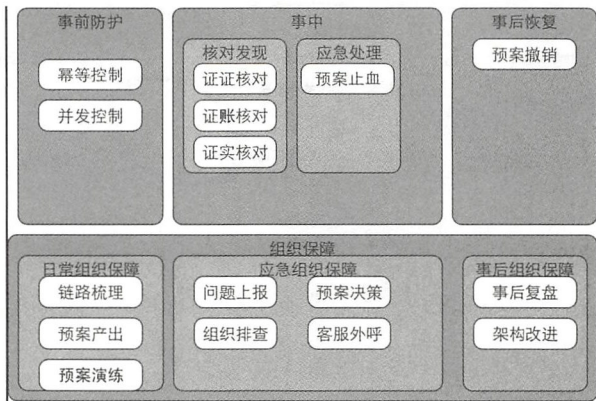


图 11-2

大促保障工作也是将复杂问题分而治之的过程，如同架构的设计过程，从功能性架构分解到非功能性架构。整体流程如下。

首先，大促保障立项，确定大促保障组织，以及负责人和各个分项负责人。

在大促的准备过程中，首先会成立大促项目，确立整体总负责人，总体负责人对整个项目负责，一般分项负责人包括业务负责人、大促应用负责人、运维负责人、数据库负责人、安全负责人、大促项目管控负责人、大促压力测试负责人、大促性能保障负责人、中间件负责人、各个业务的负责人，业务方负责人通常是业务运营代表。大促负责人负责项目进度的跟踪和问题处理，并且定期发送报告，报告进展。

其次，业务目标获取和分解，沟通业务玩法，同时确定业务数据化目标。

业务评估主要关注业务量和业务玩法，特别需要关注各种渠道的流量分配，以及业务的客单价预估，需要提醒的是不同渠道、不同商品的客单价不同。业务玩法主要是为系统的峰值容量评估提供依据，例如秒杀的玩法，往往在 10 分钟之内就能够消耗完，就不能按照一天的访问量通过二八原则（峰值是平均值的 4 倍）进行预估。

再次，容量评估和验证，根据业务玩法分解，进行容量换算。

容量换算通常从粗到细，各个负责人根据初步的容量换算指标，细化成系统的容量指标。再度量单服务器的峰值 QPS，并进行扩容采购。与此同时，对目前系统的单机容量进行度量，再将目标 QPS 和单机容量进行除法操作，换算成需要增加的机器数，一般网络工程师、数据库工程师会对容量进行评估。线上扩容之后，通过容量验证来发现影响系统水平伸缩能力的瓶颈。

与此同时，需要提前进行风险的梳理，主要针对稳定性和性能的风险，尤其是对关键链路的依赖进行梳理，将关键链路依赖的服务、存储整理清楚，同时应用负责人从应用整体出发，看是否有其他链路包含在这个应用里，这些非关键链路是否对非关键应用产生影响。网络工程师可以根据容量需求梳理网络带宽的要求和核心设备已经使用的年限情况，如果不满足大促的容量峰值要求，就要升级设备。

在压力测试过程中，要抽出部分时间进行监控点的梳理，特别是大促的监控大盘，通过一个大盘，可以看到以终为始的关键指标的情况。大盘一般分为业务大盘和技术大盘，技术大盘主要关注关键系统资源的使用情况，关注关键链路的异常数量是否有增加，一旦异常数量有增加或者关键系统资源开始超负荷（CPU、Load、Memory 等），那么要从备用机器池中进行紧急扩容，或者启动应急预案，让异常数量降下去。

最后，针对容量验证的结果，对系统的水平伸缩能力进行升级。

有些问题是代码层面的、架构层面的，有些是设备容量相关的，需要进行设备升级。如果机柜的交换机带宽过小，那么需要升级设备，如果 4 层负载均衡采用 F5 硬负载部署，并且 F5 带宽和处理能力达到上限，那么也需要升级设备。升级完成之后，需要重新对容量进行验证。

11.2 大促保障体系详解

11.2.1 容量保障体系

做好容量的保障工作，首要的是对业务运营的玩法做详细的了解，有时候业务人员对技术并不是很了解，有时候要将业务术语和系统进行适当的映射，要换算成应用、重要链路、中间件及依赖对应需要准备的容量。

1. 业务指标

为了做大促，业务上要提供目标，例如销量、用户数，还有转化率。要定一个目标，同时要给出每个细分的数据，例如上聚划算，聚划算会引流到哪个页面，大概预计多长时间内达到



目标，什么时间点发购物券，预计多长时间内被领完，领完大概多长时间内消费完，都需要预估。

业务指标要有以下 3 点信息，以便做容量评估和风险保障，同时能够识别重点保障链路。

1) 总目标量

总业务目标主要做总体目标换算使用，在电子商务类型的大促下，通常业务总目标量的指标是 GMV（交易额）、订单数，过程指标包含转化率、客单价、用户数、流量，以终为始的总体目标可以逐步换算到上游系统，例如订单数的设定目标是 1000 万单，换算成交易下单 $QPS = 1000w / (24 \times 3600) \times \text{峰值权重}$ （一般是二八原则），再往上游推，可以推算出从宝贝详情页面直接下单的 QPS，购物车下单的 QPS，所以总目标量可以做关键链路的 QPS 换算使用。仅有总目标量还不够，大促容量的整体工作需要分解，通过分解再对第一轮评估做校核，容量的评估一般需要由业务的细分玩法决定。例如业务的玩法是在某个集中的时间点进行优惠券的发放，那么在这个时间点的容量评估和按照以终为始的指标进行换算的评估要有所差异。

2) 细分目标

在进行大促合作分工的过程中，通常每个团队会将总体目标拿回去各自进行评估，然后汇总给大促项目组和机器采购人员。在各自分工的领域，需要再进行细分，确定细分到哪个端（移动端、PC 端），哪个渠道（聚划算渠道、外部引流渠道），哪些产品是重点销售的，哪些产品是爆品，目的是更加清楚地了解各个渠道不同的容量需求。

3) 运营玩法

业务玩法对容量评估起非常关键的作用。业务玩法是运营的方式，通常有预热型、秒杀型、全天营销型、分批营销型。预热型表示容量峰值可能需求较低，它提前将容量需求进行释放。在 618 期间，用户通常喜欢提前将计划购买的商品放入购物车，购物车的容量在预热期间提前释放；秒杀型通常是针对爆品而言的，秒杀型的商品通常库存和价格都非常低，用户行为会在一瞬间释放出极高的容量需求；全天营销型可以根据历史的峰值进行换算；分批营销型一般将优惠券批发送，商家和平台在大促期间根据业务目标是否达到决定是否通过发优惠券来刺激消费。

2. 系统容量评估和换算

由于大促的营销活动很多都是新业务，因此主要按照二八原则进行容量评估，不同特点的业务，峰值的倍数会有所不同，如秒杀型的玩法，需要根据预计时间内完成的销售情况进行调整，同时需要参考历史的情况。



容量评估需要评估全链路系统的容量，包括应用、服务、存储（DB）、中间件、CDN、网络、交换机、路由器、负载均衡设备等。特别要提一下 CDN 的容量评估，CDN 评估可以先根据链路的容量情况，再根据链路平均有多少图片访问，一个页面访问的 QPS 对应多少张图片或者静态资源的访问来进行。空间上，可以根据平均的图片大小计算出图片总体占用的空间，并将这个数据提交给 CDN 提供商。

3. 单机压力测试

单机压力测试主要通过测试单机的极限吞吐量和容量，以及现有的机器情况，为采购提供扩容的依据。单机压力测试通常有下面两种方式。

- 线上引流压力测试：将线上流量通过 4 层或者 7 层负载均衡设备调整机器的权重导入一台机器，直到机器的容量达到上限。要特别注意 SOA 服务的压力测试，引流时需要注意时刻观察服务的情况，因为服务通常被很多业务调用，例如交易、商品、会员服务，不能让这些公共服务将单机的容量压到极限，一旦过载，会造成大故障。
- 压力测试脚本干预压力测试：通常在大促的新页面或者有新需求时，由于业务并没有对外开放，需要人造流量，将流量导入线上，通过干预的方式，设定一定量的并发，通过混合场景压力测试度量出线上单机的容量。

4. 机器预算

根据单机压力测试结果可以评估出机器预算的汇总中间件、数据库、网络等，同时要评估 CDN 的容量。

机器采购预算如表 11-1 所示。

表 11-1

| 集群类型 | 当前机器数 | 单机容量 | 大促预计需要的容量 | 需要扩容机器的数量 | 容量评估依据 | 备 注 |
|--------------|----------|-------|------------|--------------------|-----------|--|
| 应用/中间件/服务/硬件 | 当前集群机器数量 | 单机吞吐量 | 评估大促需要的吞吐量 | 需要的机器数量减去当前已有的机器数量 | 容量评估的依据公式 | 针对容量评估的依据做出说明，扩容的公式，峰值是按照二八原则进行计算，还是按照一三比例进行评估 |

5. 容量验证

容量验证是逐步的过程，扩容完成之后，各个应用负责人会单独组织应用集群的压力测试，验证集群是否能满足大促的需求，后面要整体做全链路压力测试。全链路压力测试是完全模拟大促当天的情况，让系统的容量全部达到极限，验证整个机房的容量上限，此时中间件、网络、



公共服务的容量达到极限，由于资源之间存在互相依赖的关系，应用集群的压力测试只能验证部分容量，有些公共服务、设备的容量瓶颈并没有被探测出来，只有全链路的模拟才能确保公共服务的容量能够满足真实的需求。当然无论是哪种验证方式，在实际过程中，为了减少对业务的影响，要尽量做好关键指标的监控，同时做好应用的监控，当出现异常或者关键指标出现问题时，要立即停止压力测试。

对于同步调用，大型网站一般采取的容量验证方式有以下两种。

- 机房内部压力测试：施压机器在机房内部，直接压力测试集群的 VIP，通过 VIP 入口压力测试到对应的集群，机房内部压力测试适合于单机房压力测试，双机房压力测试通常会造成流量不均，后面会有案例进行说明。压力测试机器的成本还是比较高的，一般一台物理机器的压力测试需要 300 个并发。施压时会出现各种问题，例如文件描述符的限制问题、施压机将机柜网络打满影响压力测试效果的问题、在多活机房压力测试时 DNS 不均衡的问题。
- CDN 服务器压力测试：全链路的压力测试需要极高的容量，对于机房内部压力测试有很多的干扰和问题，比如针对双机房的压力测试，由于 DNS 的缓存问题，会造成严重的流量不均，而 CDN 压力测试通过众多的 CDN 节点，DNS 的解析结果也比较均衡，它主要模拟用户分散的特性，让流量场景更加接近用户的真实访问。

对于异步调用，要验证任务的处理能力，通常采用蓄洪压力测试的方式进行压力测试。

- 蓄洪压力测试：针对异步化的流程进行压力测试，主要针对异步消息和消费能力进行验证，同时针对异步化之后的任务表扫描、处理进行验证。将消息堆积到一定的场景下，要造异步化消息的流量，并且打标，积压到一定阶段，来看消费方的消费、处理情况是否达到预期。在实战过程中，很多业务场景对异步处理时间有一定要求，例如保险出单要求在当天完成，而业务可能要求保单的生效在第二天完成，此时任务处理事件如果比较慢，会导致业务问题。另外在保险场景下，如果保单 24 小时没有生成，机构可能会拒绝出单，造成用户投诉等问题。

在实战中，笔者经历了从线下到线上的容量验证，从单场景验证到全链路验证，从非隔离到流量隔离验证，从读链路到写链路验证的全过程。

从线下到线上的验证在业务发展的初期没有很多经验，做这个决策需要很大的勇气，因为线上验证对业务有影响，万一线上压力测试出了大故障就得不偿失了。还好好的是，其他地方已经做过线上的验证工作，只要提前做好关键指标的监控，一旦有异动就立即停止，就能确保系统没有问题。



从单场景到全链路的验证。单场景的压力测试虽然能够度量单链路和单应用相关的容量峰值，但是在大促活动期间的实际情况是，各个业务都在访问的公共资源仍然会有瓶颈出现，所以全链路压力测试是全方位、真实模拟测试集群瓶颈和系统水平扩展能力的常见方式。

从读链路到写链路，从模拟写到隔离压力测试写链路的过程，现在对于大多数场景来说，读远大于写，这和交易相关，交易系统的业务特色是用户先观看，再到决策的过程。所以写链路的瓶颈随着交易量变大而改变，压力测试通常也是从读链路到写链路过渡，随着网站规模的变大而做相应的策略调整。

11.2.2 风险保障体系

一般在业务评估完成后大促的风险保障可以并行，风险梳理越早越好。风险保障体系主要包括风险梳理、风险决策、风险开发和最重要的预案产出。

1. 风险梳理

风险梳理是根据大促保障目标进行风险初步梳理的过程，可以根据风险保障策略梳理出风险。一般风险梳理会根据应用的级别将应用的关键链路和关键链路的直接依赖梳理出来。根据稳定性公式可以看到一个链路的稳定性和依赖相关，所以风险梳理的重点是将关键链路的直接依赖先梳理出来，然后假设这个依赖出现宕机或者服务不可用，观察会对链路造成何种影响，是否会造成用户打不开页面的情况。同时要从应用的整体维度看，哪些链路占用了应用的大部分访问，哪些非关键链路会严重影响关键链路。

风险梳理的链路一般从用户访问的视角进行整理，例如对于电子商务业务来说，链路包括首页、搜索、宝贝详情、购物车、下单、确认收货和订单查看。在大促时，需要新增链路，包括主会场、下单、确认收货和订单查看。

注意链路的梳理是关注点分离、分工进行梳理的过程，应用工程师可以从应用的视角进行链路的梳理，而网络工程师的视角是从网络入口到机柜的服务器，CDN 工程师的视角是从 Edge Server 到回源站的链路，收集图片、JS、CSS 等静态资源的访问量，反馈给 CDN 服务提供商，并且要求提供商参与大促的准备。

笔者从依赖这些链路的应用维度将影响应用的非关键链路梳理出来，举例如表 11-2 所示。

2. 风险决策

风险决策指的是根据风险的优先级和链路的重要性来决策，哪些风险是需要重点保障的，哪些风险是确定要重点做改进方案的。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

表 11-2

| 应 用 | 链 路 | 直接依赖 | 风险假设 | 风险类型 | 风险优先级 | 是否是新链路 | 备 注 |
|-------|---------|---------|-------------------------------------|-------|-------|--------|-----------------------------------|
| 归属的应用 | 关键链路 | 指定依赖是哪个 | 通常是假定直接依赖不可用，是否造成这个链路不可用 | 风险的类型 | 风险优先级 | 是否是新链路 | 需要特殊说明的地方 |
| 购物车应用 | 购物车商品列表 | 卖家黑名单服务 | 如果挂掉，由于程序做了异常处理保护，会进行自动降级，不做黑名单校验 | 稳定性风险 | 低 | 非新增链路 | 需要特别考虑容量，考虑超时时间是否设置合理，建议设置较小的超时时间 |
| | | 店铺服务 | 如果服务不可用，购物车列表无法展示 | 稳定性风险 | 中 | 非新增链路 | 设置合理的超时时间，因为店铺服务是公有服务，考虑容量报备 |
| | | 会员服务 | 如果服务不可用，由于会员服务存在双链路，所以不会导致购物车列表无法展示 | 稳定性风险 | 非新增链路 | 中 | 设置合理的超时时间，因为会员服务是公有服务，考虑容量报备 |
| | | 购物车DB | 如果DB不可用，购物车无法展示 | 稳定性风险 | 非新增链路 | 高 | 设置合理的超时时间，需要重点做保障 |

3. 风险改进

风险决策完成之后，需要制定对应的改进策略，对风险制定改进方案，风险改进方案可能会有不小的开发工作量。风险改进策略有很多，有一些比较好的模式可以很好地应对常规的风险。风险改进不仅要制定策略，也要制定改进计划，举例如表 11-3 所示。

表 11-3

| 应 用 | 关键链路 | 风 险 | 风险改进方案 | 预计上线时间 | 负责人 | 备 注 |
|-------|---------|---------------|------------------------------|--------------|-----|-----------|
| 归属应用 | 链路名称 | 风险描述 | 针对风险的改进方案，可以链接到详细的方案设计文档或者页面 | 风险改进预案的上线时间点 | 责任人 | 需要特殊说明的地方 |
| 购物车应用 | 购物车展示页面 | 购物车数据库容量不足的风险 | 限流方案，设置合理的限流值 | | | |

4. 预案产出

针对风险，最终产出的预案列表举例如表 11-4 所示。预案要有明确的风险定义和预案的执行条件，通常通过监控发现问题后进行预案的执行。预案分为常规预案和应急预案，常规预案一般是大促开始前执行，应急预案是为应对意外而产生的预案。常规预案一般是根据业务制定



的，对业务无损伤，应急预案可能会对业务和体验有影响。例如 618 期间，在零点，为了确保下单链路无影响，用户在短时间内无法查看订单列表。

表 11-4

| 业 务 | 链 路 | 风 险 | 预 案 | 预案类型 | 预案执行条件 | 预案执行步骤 |
|--------|--------|----------|------|--|-----------------------|--|
| 归属的业务线 | 关键链路名称 | | 预案名称 | 常规预案：大促开始前就开始执行； 应急预案：在大促发生某种情况时启用该预案 | 预案的执行条件，发生什么情况，就执行该预案 | 预案的执行步骤，包括预案什么时候关闭，主要预案可能会影响用户体验，例如在618期间，由于商品库的压力过大，可能会有订单列表关闭预案，用户在此期间不能查看订单 |
| 宝贝详情 | 宝贝详情页面 | 物流运费容量风险 | 降级 | | | |

11.2.3 组织保障

组织保障是大促保证的重要一环，要保证大促的顺利进行，组织上不仅要做资源的倾斜，并且要有一定的授权，同时各方达成一致。

大促组织保障包括大促准备阶段的组织保障和大促当天的组织保障。大促当天的组织保障主要是大促指挥部，所有的大促预案在大促指挥部的指令下才能执行，大促指挥部也做其他的重要决策。大促组织包含大促指挥部、下属各个分业务组织。在发生重大问题时，由各个小组进行问题汇总上报，由决策委员会进行决策，各个小组分工盯数据大盘，随时应对发生的问题。大促组织结构如图 11-3 所示。

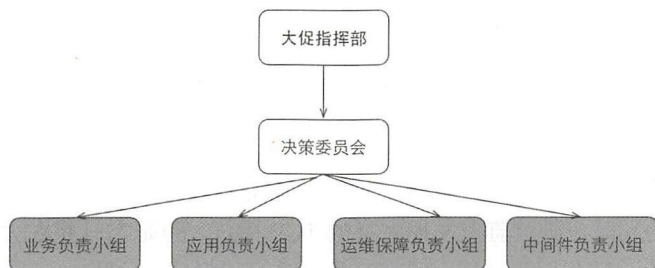


图 11-3

11.2.4 运维保障

运维保障是指从机房入口到机柜的机器的全链路保障，流量从入口到核心路由器，到核心



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

交换机，再到机柜，运维主要负责网络设备、机柜的更新和安全保障工作。在大促期间为了防止恶意攻击，要做很多安全上的保障，常见的包括 3 层攻击和 7 层攻击防护，7 层攻击防护主要是应用服务器（如 Nginx）的防护策略。

1. 机房容量保障

从机房的维度来说，电量大小和所能容纳服务器的数量、机柜的数量都是有限的，在大促准备的时候，需要考虑机房是否能够扩容，如果机房容量不够，需要考虑同城机房。同城机房的建造需要花大量时间，包括机器和各种设备的采购、机房的选址、架构的部署和搭建、业务测试和回归测试、灰度验证等。机房级别的改造和升级，需要提前至少半年到一年。同城多活的机房架构，一般采用双机房热备的方式，机房间可以互为备份，流量比例可以任意切换。

2. 网络容量保障

大促保障期间，业务的发展情况不同。在初期，为了节约成本，设备通常会采用千兆网络，核心交换机和机柜的上联交换机都是千兆设备。当业务发展比较快的时候，网络容量非常容易出现瓶颈，特别是图片回源服务器是非常占用带宽的，而且在网络部署规划上，由于初期业务量小，会简单地将大流量的应用和小流量的应用进行混合机柜部署，这些应用会大量占用带宽，小流量应用是和大流量的带宽进行共享的，从而导致非常容易出现网络瓶颈。

针对网络保障，总结如下。

（1）梳理网络部署：提前对网络的部署进行梳理，重点对关键链路的机柜、服务器带宽进行梳理，为后续的升级提供最基本的素材。

（2）隔离部署优化：针对大小流量系统进行隔离部署，关键链路和非关键链路隔离部署，关键链路之间也需要做适当的隔离。

（3）升级网络带宽：在预算允许的情况下进行带宽的升级，如果预算有限，可以考虑将关键链路所在的机柜带宽进行升级。

在某次大促实战中，由于机房比较老，机柜上联交换机设备陈旧，带宽容量低，在压力测试的时候发现大量的超时，通过监控发现大量的 TCP 重传，但是当时是在大促准备过程中，没有提前购买网络设备，只有少量机柜上联交换机带宽可以升级。

11.2.5 中间件保障

中间件包括 RPC 中间件、缓存中间件、消息中间件、分库分表中间件、存储解决方案和同步中间件。中间件通常需要做到较高的伸缩性，只要增加机器就可以解决容量的问题，但是中



间件和存储同样存在容量上限。

11.3 大促容量峰值保障策略

在大促过程中，系统需要 4 大能力，这 4 大能力是峰值保障的基本点，如图 11-4 所示。

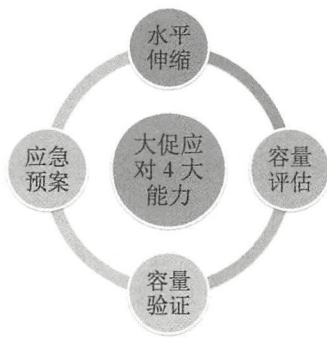


图 11-4

应对大促的时候，上面 4 大能力，对大促的成败起至关重要的作用。

- 容量评估能力：比较精确的容量评估能力是非常关键的，如果能够预估每个关键链路和关键业务的容量需求，那么可以根据预估的容量通过扩容来进行容量的线性增加。
- 容量验证能力：机器扩容之后，需要对集群进行吞吐能力验证。
- 水平伸缩能力：水平伸缩指的是通过线性增加机器来提升系统的吞吐能力，大促时瞬间流量大，对系统的峰值处理能力有很高的要求。大促保障很重要的一个工作是保障峰值的处理能力，确保系统能够轻松应对瞬间的峰值。从单个机房来说，要从流量流向的链路逐层分析，从 2 层、3 层的路由器、交换机、机柜、服务器，4 层的软负载均衡 LVS（F5），再到 7 层的应用层，都需要具备水平伸缩能力。从机房级别来说，架构和部署上要既支持同城双机房，又支持异地多活多机房，这些架构是水平伸缩能力的关键。
- 应急预案能力：在发生问题时，能够通过提前准备好的应急预案解决问题。一般是通过梳理和容量验证之后，发现可疑的风险点，经过预案的开发沉淀出应急预案。预案分为常规预案和应急预案，应急预案在大促当天进行操作，有明确的发生条件，因此针对预案需要有完善、完整的监控部署，才能快速地发现问题，同时通过细分化的监控来完善快速分析能力，为快速解决问题提供时间。

围绕这 4 大能力，大促峰值保障策略总结如下。



- 做好容量规划，清晰掌握容量需求：通过业务目标到系统容量的换算，计算出大促的容量需求，并提前进行采购扩容，同时为风险梳理提供数据参考。在风险梳理时，可以对现有的峰值能力和目标峰值能力进行比较，能力差距大的峰值可以列为风险。
- 度量每个关键链路的现有峰值能力：对每个关键链路的现有容量通过压力测试进行度量，清晰地了解现有的峰值处理能力，便于及时改进和升级。
- 梳理每个容量风险：根据容量规划计算出明显的容量瓶颈，包括设备的网络带宽和存储空间，特别需要重点关注单点、单数据库的风险。笔者多次经历过单数据库引起的容量瓶颈造成短时间不可用，针对单点容量瓶颈需要提前进行规划和改进，不要等到容量验证后才开始改造。
- 对公共依赖进行上报：对公共依赖的容量需求进行报备，以免被限流。大促期间公共服务如交易、店铺、会员，还有中间件的容量需求要上报，各个服务的提供方会根据这些容量进行保障，如果没有报备，可能会进入限流池。
- 提前做水平伸缩能力的改造：一般重大的架构改造需要很长时间，水平伸缩能力需要提前建设，识别到容量风险就立即行动，特别是关键链路，越早做改造，大促保障工作就越轻松。
- 通过压力测试进行容量的验证：通过多轮、分层的压力测试来检验集群的峰值吞吐能力是否符合大促的容量需求，同时对不满足需求的系统进行整改和升级。压力测试工作应该分层次进行，提早进行单集群压力测试、单场景压力测试，再进行混合场景压力测试，最后进行全链路压力测试。经过分层压力测试，可以提前发现系统的伸缩性能力问题。
- 对公共依赖预案提前了解，并评估对业务的影响：公共服务也会做各种预案，有些预案是对业务有损伤的，最常见的预案是限流，例如转账业务可能会异步化，异步化之后转账可能会延迟，需要提前获取这些预案并采取对应的措施。针对转账延迟问题，可以在产品设计时，将文案通告提前挂在网页上进行告知。笔者曾经经历过，在大促期间由于确认收货服务降级，导致用户无法确认收货，后续流程无法继续，引起了用户的投诉。
- 做好链路的分析和监控，快速发现瓶颈：监控是发现问题和分析问题的基础，一个好的监控系统不仅可以更快地发现问题，也能更好地分析问题。
- 对关键链路做好预案：对容量的风险做各种预案，包括技术预案、业务预案，提前做好详尽的预案，并且一定要对预案进行演练。
- 流量控制兜底应对突发流量：流量控制是峰值保障最通用的方案，确保系统不会因为突发流量崩溃。
- 备用机器，应对容量预估误差：容量评估和度量可能存在各种误差，可以通过预留一部分机器（10%）来解决问题，它不像流量控制会对业务有影响。

11.4 大促风险保障策略

11.4.1 风险保障概述

大促保障是“容量保障+风险保障”的过程，大促保障首先要满足在促销过程中产生流量突变带来的容量需求，面对容量需求，通过一定的度量手段，确保技术能够满足业务的需求。大促保障的过程又是风险保障的过程，是风险的全面梳理、识别、改进和针对风险进行（应急）预案的过程，主要保障稳定性。

大促保障是保 3 样东西，保峰值、保关键业务、保关键链路，保峰值很容易理解，无非是满足业务的需求，保关键业务、保关键链路无非是保障和大促相关的链路。但在具体实施的过程中，在此基础上要做一些额外的考虑，笔者曾经遇到一个由于旁路（非关键业务）风险导致关键业务受到重大影响的案例。

11.4.2 风险保障常见风险

- 全局业务保障难度大
 - 大促一般伴随着业务流量突变，所以一些公共资源风险保障难度增大，公共服务如 IC、TC、支付这样的链路每个业务都有容量需求，因此保障难度增大很多。
- 流量突变峰值需求高
 - 大促期间有运营拉新带来的渠道流量冲击，对系统容量要求更高，流量带有突发性，日常的流量上涨更加平滑，流量突变对峰值的要求更高。
- 安全性风险差异大
 - 由于大促活动带来各种交互的变化，这些新的项目 and 需求会带来更大的旁路风险。
- 稳定性要求高
 - 稳定性公式如图 11-5 所示。

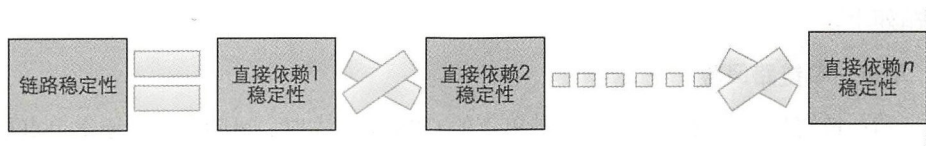


图 11-5

11.4.3 风险识别和风险分析

在大促的准备过程中，会收到大促应急小组要求的风险梳理流程，什么是风险？这里的风险主要指技术风险，不是大促相关的项目延迟风险、运营关键流量部署风险，所以要将链路中的风险识别出来。

根据稳定性公式可以得出，只要依赖链路的风险提升，关键链路的风险就会提升。

(1) 根据稳定性公式，风险和依赖直接相关，所以风险的识别主要围绕着依赖进行，有下列基本原则。

① 识别依赖优先原则

- 首先要确定关键链路有哪些依赖，这些依赖有的是直接依赖，有的是间接依赖，都需要梳理出来，依赖梳理是风险识别的前提。
- 要对关键链路所承载的应用进行完整的梳理，包括非关键链路的梳理，也就是下面要讲的旁路风险，通常如果非关键链路的调用量和调用次数远超关键链路，那么旁路风险会更加明显，所以根据识别依赖优先原则，要从整个应用出发进行梳理，而不只是梳理业务上的关键链路。

② 依赖故障假设原则

假定依赖的服务或者中间件出现无法提供服务的问题，链路会有怎样的表现，是否会造成关键链路不可用，是部分不可用还是完全不可用？根据影响的范围能够确定风险的等级，对于页面来说，是不是会出现页面无法展示的情况，如果会，那么这种链路就存在稳定性问题，需要考虑应对风险的方法，也就是后面将要提到的风险保障策略。

(2) 大促主要做的是容量保障和风险保障，链路的稳定性和直接依赖密切相关，所以大促的风险也围绕容量和稳定性风险展开。所有有碍容量和稳定性的都是风险。

回想以前做过的大促和最近梳理风险过程中遇到的问题，再加上依赖故障假设原则，风险分类总结如下。

1. 单点、单链路风险

单点和单链路风险等级最高，特别是单点的 QPS 不太大时，风险更大。通常单点稳定性提升，需要使用多冗余策略，同时与分布式风险分散策略搭配使用，通过自动冗余路由，在单链路出现问题时，通过另外一个链路自动启用或者应急启用来防止单点不可用问题。

常见的单点和单链路风险如下。

- DB 单库风险
 - DB 单库风险主要来自容量不足造成的连接池满，一旦在大促期间出现问题，由于流量持续处在高位，会出现较大的故障。
 - DB 单库风险还来自单库服务器出现问题时，风险无法分散导致全部崩溃。
- 单机房风险
 - 单机房风险通常出现在一些特殊情况下，例如单机房的某个硬件设备坏死，包括交换机老化、路由器老化、网络线路坏死、机房特殊断电、机房电量供应不足等，都会出现单机房整体故障，此时可以通过双机房冗余策略来提升稳定性。
- 热点链路风险
 - 通常一个分布式系统具有一定的抗不可用能力，但是当出现热点时，热点会被分配到一个单点上，此时就会造成故障。针对热点的策略，或者通过近端策略来减少热点的压力，或者通过业务上缓存架构的重新设计来减少热点的问题。

2. 旁路风险

旁路风险指的是非关键业务对关键业务的影响，因为非关键业务和关键业务可能会共用资源，而非关键业务直接影响关键链路。这个风险很容易被人忽视，通常如果旁路直接调用数据库，并且调用量特别大，会直接将关键业务的资源耗尽。笔者曾经历过两次旁路风险，一次是大促开始前 40 分钟，因为旁路调用量过大造成关键业务受损。旁路风险一般通过旁路熔断降级、旁路限流和旁路分散策略进行应对。在一次 618 大促过程中，在梳理某个系统的风险时，通过流量构成梳理发现，关键链路的调用一天才 100 万次，但是旁路的调用次数达到 5000 万次，这个旁路如果出现抖动对关键链路的影响非常大，因为关键链路和旁路在一个应用中进行处理，CPU 资源和线程池资源都是共享的。

3. 容量风险

业务需要的容量和实际容量存在的差距，叫作容量风险。容量风险一部分通过梳理获得，一部分通过全链路压力测试进行检验。预测容量风险需要获取每一个依赖的容量上限。

- 网络设备的容量上限
 - 万兆网卡打满，可导致整个大促容量无法支撑。
 - 源图片服务器和应用混在一起，由于图片服务器占用了大量的交换机带宽资源，导致在同一机柜中的机器受到影响。
 - F5 最大带宽上限为 10GB。
 - 安全攻击防护设备存在明显带宽上限（例如 80GB）。

- 连接上限
 - Oracle 的连接数上限为 3000，设定后如果要修改，需要重启服务器，单库的 QPS 上限为 2000~4000。
- 存储上限
 - 单表数据量上限为 2000 万~4000 万。
 - 搜索引擎存储上限与服务器的行列数相关，行数代表 QPS，行数越多 QPS 越高，列数表示索引的数量。
- 公共服务上限
 - 618 期间，一些公共服务都会有限流，没有报备的服务非常可能在限流之列。
 - Tair 单机的 QPS 大约在 10 万~20 万，热点的 QPS 不超过 5 万。

4. 雪崩风险

随着访问量的增加和持续，一个单点故障会加重资源的消耗，导致所有的请求都受到影响。笔者曾经遇到一个场景，由于 Dubbo 有自动的三次重试，当系统出现问题时会不断地重试，从应用端进行压力测试只有 200 个并发，但是服务提供方却有 600 个并发，导致服务完全不可用。

另外一个风险和超时时间设置相关，超时时间太长，会造成线程资源不能快速被释放，从而导致所有的连接都受到影响。

5. 非相关共享风险

共享有很多含义，其实任何链路之间都存在某种共享，例如一个应用中所有的 CPU 资源和内存资源都是共享的，一个机柜内所有的网络带宽都是共享的。共享风险不是指这些风险，而是指完全不同的关键链路由于共享某些资源，让看似不相关的资源产生风险。最典型的是 HSF 连接池共享风险和数据库连接池风险。

6. 级联冗余风险

级联冗余风险通常指的是，在双链路情况下，由于主链路出现问题，另外一个链路无法承受大流量冲击，从而造成链路不可用。通常发生在“集中式缓存（如 Tair）+DB”的经典双链路搭配时，Tair 由于容量远大于 DB，当 Tair 被击穿或者 Tair 稍微抖动时部分击穿，DB 承受不住流量冲击，导致链路不可用，DB 此时变成强依赖，从而导致级联冗余风险。级联冗余风险通常的应对策略是“熔断非优先调用+优先保障”，在流量非常大时，打到 DB 实际上是不可行的，因此对于多个链路，要保障 DB 给那些更加需要的地方、业务优先级更高的地方。

7. 强弱倒置风险

通常在大促风险梳理的过程中，会要求梳理出强弱依赖，但是很多时候本应该是弱依赖，

由于依赖处理存在问题，变成了强依赖。这种强弱倒置的问题经常发生，例如购物时有个获取卖家黑名单的逻辑，由于开发人员没有正确处理异常，在卖家黑名单服务出现问题时，导致整个购物车页面不可用。这种强弱倒置的风险，在日常保障和大促保障时都需要关注，在某次平台风险梳理的过程中，笔者发现当调用类目的服务出现问题时，整个搜索页面无法打开，从而导致用户购买链路受到影响，所以针对强弱倒置风险，进行了风险应对。

11.4.4 风险保障策略

1. 全链路压力测试策略

全链路压力测试是发现问题最直接的手段，通过链路的全覆盖，能够发现大部分的中间件、应用、服务、软件、硬件的容量瓶颈，风险也更加一目了然。全链路压力测试是以每个系统业务需要的峰值吞吐量为基本目标的，没达到目标则需要改进。全链路压力测试最大的问题是链路的覆盖是否完全，要真正做到全链路，需要对链路上所有业务的流量进行盘点和梳理，这是非常浩大的工程。全链路压力测试的成本是非常高的。通过全链路压力测试发现了风险，不一定能够及时修改，所以全链路压力测试需要和风险的梳理及识别结合进行，才能做到万无一失。

2. 减少直接依赖策略

优先保障关键链路和强依赖，故障隔离策略包含异步化解耦策略、优雅降级策略、熔断保障策略，将直接依赖变成间接依赖，减少直接依赖。每减少一个依赖，系统的稳定性会提高一个档次。下列常见的稳定性保障策略都是减少直接依赖的策略。

1) 强依赖变弱依赖策略

强依赖越多，链路的可用性越差，在编写代码过程中，会存在各种问题。有些工程师在编写代码阶段不知道什么时候用强依赖，统一全部使用了强依赖的方式进行关键链路的设计。

2) 近端策略

近端策略在大促风险应对策略中是常见的方式。所谓近端策略是指，将长链路依赖变成放在近端执行的方式。近端策略有效地减小了由于直接依赖的不稳定或者容量不足造成的风险。

例如，在某次大促的风险梳理过程中，发现 Tair 抖动会引起商品详情页的不可用，针对这个风险，笔者准备了 Tair 近端缓存策略，将商品详情的远程 Tair 缓存改成近端缓存的方式，并且仅在大促当天生效。例如，618 下单确认页面会调用某个业务的推荐服务，618 期间淘宝商品详情页的调用量非常大，并且存在不可预知的流量突发风险，考虑到这个风险，使用二方库 jar 包将推荐服务的远程执行改成在商品详情页所在应用的本地来执行，以减少不可靠的链路风险。

近端策略是通过减少直接依赖来进行风险防护的常见手段。

3) 异步化策略

异步化策略是常见的稳定性保障策略之一，异步化策略可将链路的依赖由直接依赖变成间接依赖。

例如，在某次风险梳理过程中，笔者发现第三方合作保险机构存在极大的容量风险，大促系统和第三方机构系统是直接连接的。在保险购买的过程中，一般先让用户去保险公司核保，从机构获取投保订单号，核保通过后才进行支付。针对这个风险，最初有两个方案，第一个方案是让机构同意进行核保前置忽略，也就是在大促当天，与机构进行沟通，让用户购买保险产品时不核保，但是在短期内通过全链路压力测试和容量测试让机构进行改进的想法很难实现，考虑到这个风险，决定采用异步化策略进行核保动作，完全解除在关键链路上对架构的依赖，异步化策略可以有效地进行容量的削峰填谷，让容量变得更加平滑。

4) 熔断降级策略

熔断包含全局熔断和链路熔断。当一个公用资源受到影响时，根据链路的优先级，将非关键链路进行熔断降级。例如，在淘宝 618 时，当公用下单写入和读取服务出现问题时，熔断订单列表查看功能，等峰值过去再恢复。

5) 多冗余策略

多冗余策略是指，当某些依赖出现单点故障时，启用另外一个链路的策略。多冗余策略是大幅降低风险的主要措施之一。

例如，一个链路的可用性是 99%，另外一个替换链路的可用性也是 99%，那么双链路的可用性 $= 100\% - (1\% \times 1\%) = 99.99\%$ ，双冗余链路可以让稳定性提升 0.99%，收益非常可观。

6) 风险分散策略

常见的分布式系统采用的是典型的风险分散策略，在实际风险梳理和大促保障时，经常会遇到单库由于容量问题产生的风险，将数据库进行水平、垂直切分，可以有效地分摊风险。风险分散策略主要应对单点和单链路风险。

7) 限流过载保护策略

限流过载保护策略是保障超设计容量的策略。当实际流量超过保障范围时，系统由于资源的消耗，可能所有用户都不能访问。针对这种场景，一般都通过限流策略进行过载保护，过载保护可以确保在阈值内的用户正常访问，如果不限流，所有用户可能都不能访问。限流过载保护策略实际上也是一种优先保障策略。

11.4.5 分组隔离策略

对于特别需要保障的链路，可以将服务进行分组部署，这样不会因为非重点保障的链路影响重点保障的链路。例如，通常在用户购买流程中将链路依赖单独划分为一个组，将用户订单后台的操作分成另外一个组。

通常分组隔离包含以下两种隔离方法。

- 客户端隔离：主要避免性能差的服务影响性能好的服务，以及避免互相影响。典型的场景是 HSF 客户端连接池默认是公用的，一个复杂的系统，会依赖少则十几个服务端，多则几十个服务端，这些服务依赖放在一起共享一个 HSF 线程池，势必导致一个服务出现问题拖死其他服务的调用。客户端隔离方案一般可以根据耗时进行分组，将耗时特别长和服务风险大的线程池进行单独部署。
- 服务端隔离：服务端隔离一般采用分组的方式，对服务部署进行分组，分别提供给重要的业务和非关键业务，这样也起到了一定的隔离作用。服务端隔离一般从服务提供方的角度来防止非关键链路的服务负载影响关键链路的服务负载。

11.4.6 业务降级策略

业务降级是常用的大促保障策略，在梳理链路时，如果发现某个链路有风险，对用户体验影响比较小，可以和业务沟通，在大促期间对此业务不进行服务的提供。例如在执行保障策略时，将碎屏险的激活验机流程在 618 期间关闭，用户当天只是进行购买行为，激活放在 618 峰值之后，以便在 618 期间保障购买流程。

11.4.7 监控发现策略

监控的目的是快速发现和定位问题。由于大促期间，很多新的业务需求会上线，针对新的业务，日常监控往往覆盖不足，同时很多应急策略和方案依赖于监控。针对监控，首先还是要梳理重要的监控项，从业务监控大盘到应用监控大盘，再到系统监控大盘，应急方案是根据监控的结果来执行的，通常监控发现策略不仅要关键的监控加上，必要时还要将数据的抓取放在监控发现策略里。

11.5 大促资金安全保障策略

11.5.1 常见的资金安全防护策略

资金安全针对的是由于程序代码问题或者业务需求理解发生错误而导致客户或者服务提供

商出现资金损失的场景，不是指用户银行卡被盗刷或者手机被盗取等欺诈的场景。通常资金安全问题容易发生在并发控制、幂等控制出现问题的时候，其后果是客户少付或多付。

在实战中有很多资损发生的情况，业务需求理解错误也时有发生。如笔者曾经历的一次资损，本来产品的售价是与份数相关的，开发人员理解成价格不与份数相关，用户买 10 份的价格和 1 份的价格是相同的。

并发控制是资金安全问题无法完全避免的根本原因，只要是人设计的系统，都会有资损的风险，特别是涉及非常复杂的系统之间的交互时。例如，在实战过程中，上游系统和下游系统，涉及金额单位的问题，上游系统默认的单位是元，下游是资金处理系统，默认的单位是分，此时会有一百倍资损问题发生。又例如，在代扣场景下，一般会将会代扣的任务记录在任务表里，在代扣日到来之前进行资金代扣的处理。代扣任务的生成，同样要符合幂等原则，即确保每月的代扣任务记录只有一条，这一条记录通常通过幂等字段进行控制，在生成任务记录时，一般会有业务单据号和类型两个字段，这两个字段作为联合业务主键，确保代扣只处理一次。

常见的资金安全防护分为事前、事中、事后 3 个阶段，实行立体式防护，如图 11-6 所示。

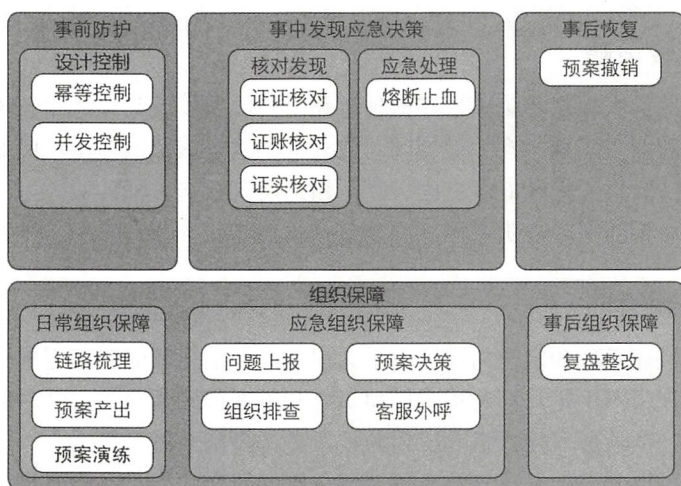


图 11-6

(1) 事前避免：侧重在事前提前发现问题，事前可以做很多事情进行资金安全防护。

- 设计层面：需要严格考虑并发和幂等控制，通常在要求开发工程师进行资金安全设计的时候，进行并发控制的设计和幂等控制的设计，幂等控制要求明确指出幂等字段，确保一笔交易只处理一次。

- 组织层面：定期对资金链路进行梳理，重点梳理核对点和核对规则，同时根据核对发现的问题，对预案进行设计开发，提前规避问题。预案通常包括熔断处理，防止资金损失扩大化。同时进行故障演练，通过故障注入的方式验证核对发现的能力，并考察开发工程师对资金安全的敏感度。

(2) 事中应急决策：事中主要侧重于快速发现问题、应急处理和快速修复。

- 执行层面：在发现问题后，立即将出问题的链路进行熔断，防止资金问题继续发生，这种措施称为熔断止血。在熔断止血期间，可以留时间进行资损问题的处理，很多资损无法通过代码回滚来解决，因为数据无法回滚。问题主要通过核对来发现，核对通常包含 3 种形式，证证核对指的是将业务单据和下游的业务单据进行核对，证账核对指的是将业务单据凭证和账务产生的账单进行核对，证实核对指的是将业务单据凭证和实际的金额进行核对，核对内容主要包括应收应付的金额核对和业务规则的核对。业务规则核对是金额核对的有效补充。在很多情况下，幂等字段值会出现问题（幂等失效）。例如，由于系统升级，开发人员修改了幂等值的构造方式，导致幂等控制失效，但是每一笔明晰账金额核对都是没有问题的，此时规则核对能起到一定的作用。针对秒杀商品，每个用户限制购买一个，此时可以在核对上增添每个用户限制购买一个的规则，这样能够发现金额核对不能覆盖的场景。通常核对时需要考虑覆盖率，所谓的覆盖率指的是目标表的记录数与 SQL 作用的记录数的比例，只有达到 100%，才能确保核对无遗漏。
- 组织层面：发现问题后，应该立即上报，并由应急小组来决定预案是否执行，预案执行完成止血动作后，组织上要立即安排问题的排查工作，同时如果资金已经流入客户账户，需要客服进行外呼追款，在客户同意的情况下，进行资金款项划回，这可能会持续很久。

(3) 事后复盘整改：事后主要侧重重新分析问题的根本原因并进行整改，将同类型的问题都一并解决。

资金安全问题的发生往往都不是表面上所看到的这么简单，需要深挖背后的原因，找到根本问题，再进行整改，这样才能彻底解决问题。笔者曾经历过一次资损，发生在与第三方外部系统进行交互的过程中，由于经常无法有效地访问第三方系统，用户经常投诉，后来开发人员为了避免这种问题，自己开发了一个工具，进行手工触发，但是涉及资金的问题，绕过了严格考核的流程，导致用户可能多拿钱。经过严格的分析，笔者将重点放在第三方系统无法交互的架构改进上，将重试数次就停止策略改成永远重试策略，同时关闭工具，从架构上彻底解决了问题。



11.5.2 大促资金安全防护

大促期间有不少新增的链路，这些链路的资金安全防护要特别注意，同时由于访问量会在瞬间产生，大量的资金处理在一瞬间开始爆发，要能够很快发现资金安全问题，这对核对的实时性有很高的要求。

(1) 有新增链路：大促期间有不少需求是新增的，所以要考虑新增链路原有的防护策略是否能覆盖，包括熔断止血、核对覆盖及幂等和并发控制的设计均需要进行全面的梳理和备案。

(2) 实时性要求高：对资损发现的问题有实时性的要求，因为大促涉及资金量大，同时考虑到有新增链路，所以对核对能力有很高的要求。在实战中，笔者做了很多不同的尝试，传统的核对一般只能达到 $T+H$ 的核对能力，而在大促期间，由于访问量大，资金规模也大，所以对核对的实时性有很高的要求，通过离线日志进行核对，很难做到实时，为了确保资金没有损失，可以通过写脚本的方式将线上的日志进行统计处理并发现问题。

11.6 大促经验沉淀

通常在大促中，有很多问题和预想的不同，一方面要随时截图，将场景记录下来，另一方面要将日志存档，作为下一次的全链路压力测试日志。在大促结束后，不仅要业务结果进行复盘，也要对保障结果进行复盘，特别是峰值曲线和平时的峰值 QPS 的关系要记录下来。在很多时候，监控系统可能只保存半年的数据，这些曲线也要记录下来，为下一次的大促提供评估依据，同时为下一次大促避免同样的问题提供经验。

1. 关键指标和数据沉淀（表 11-5）

表 11-5

| 业务 | 应用 | 峰值QPS | 应用指标曲线 | 系统指标 | 应用性能指标 | 日志存档地址 |
|-----|-----|-----------------------|---------------------|---------------------------------|--------|--------|
| 业务名 | 应用名 | 峰值QPS的数字，评估值，主要看评估的误差 | 大促当天的表现，包括QPS、RT、异常 | 系统指标曲线，包括CPU、Load、TCPretr、网卡I/O | 指标名 | 地址 |

2. 关键问题记录和沉淀

在大促期间，可能有一些没有保障到的点。例如，笔者曾经历过大促当天数据库连接池报警、营销商品无法下单的情况，当时多个数据库出现连接池不够用报警，由于单库和其他库部署在一个物理机器上，非关键业务的数据库有批量查询动作，这个查询十分耗时，同时磁盘是



共享的，导致关键业务也受到影响。

3. 经验沉淀和总结

在大促准备过程中，可以将经验沉淀下来，包括风险的排查方法、容量的验证方法，以及未来流量更高时，系统可能存在的新瓶颈和准备的策略规划，有些水平伸缩能力的准备时间需要很久。

11.7 大促保障实战分析

11.7.1 机房网络瓶颈问题分析

1. 问题摘要

为了评估容量和机器数，以方便水平扩展，传统的方法有线上引流压力测试、线下脚本压力测试和线上脚本补充流量压力测试，这些方法是通过对单机的极限 QPS 和目标总 QPS 进行评估来确定需要多少机器的，单机压力测试只能发现单机的性能瓶颈，主要作用在容量评估本身。单机压力测试需要保证各个系统没有其他瓶颈，而在实际用户访问过程中，用户对整个集群进行访问，也就是说压力在整个集群上，整个集群是否存在瓶颈通过单机压力测试很难发现。集群压力测试就是为了发现整个集群的瓶颈。

总的来说，单机压力测试的出发点是评估单机的容量，作为机器扩容的依据，同时也能发现单机的瓶颈。集群压力测试是在单机压力测试的基础上验证是否能提供预想中的 QPS 服务能力（单机×机器数），当明显不能时，说明集群存在瓶颈。全链路压力测试模拟真实用户的实际流量对系统进行访问，看是否存在增加机器解决不了的问题。

2. 全链路集群的瓶颈问题定位

1) 木桶理论是瓶颈问题定位的理论依据

要想知道用户在实际访问过程中，集群是否能提供正常的服务能力，可以通过集群压力测试把系统的问题找出来。性能优化的一条很基本的原则是，所有的性能优化都符合木桶理论：一个水桶无论有多高，它盛水的高度取决于其中最短的那块木板。一个集群的服务能力有多高，取决于依赖方服务能力最弱的系统。

2) 通过差异对比来发现集群瓶颈

经过数十次故障排查，并借鉴压力测试的经验，以及通过集群中服务器的差异表现对比能



够发现集群的某些瓶颈。差异对比是符合木桶理论的，表现有差异的系统和机器（虚拟机）往往是某些资源瓶颈的标志，而这个出现瓶颈的资源其实就是木桶理论中最短的那块木板。差异化对比符合自然界的规律——完全对等的系统理论上是不存在的，这个不对等有人为因素，如同一应用硬件配置不同；也有客观因素，如同一应用所接收的流量不同；部署在不同的机架上，网络流量有差异，交换机网卡的配置不同，机架里面的机器不对等，造成流量等不同。

3) 差异对比的操作步骤

首先查看同一个应用中 RT 的变化，找出 RT 变化大的应用。

其次分析 RT 变化大的机器，看它们的共同特性，例如网络配置、机器配置等，分析常见的资源瓶颈点。配置不同、网络流量不同、内存不同、硬盘不同、连接到后端的服务表现不同，通过分析一般都能找到瓶颈。

3. 常见的资源瓶颈

同一个应用有问题的机器表现出异常大的 RT，通常是网络出现瓶颈的标志，同时因为硬件配置不同，RT 表现也会不同。RT 和 QPS 是结果，而 RT 和 QPS 表现是否一样取决于资源，这些资源的表现是性能问题的原因，由结果到原因可以定位出是哪方面资源出现了问题，这样可以有针对性地排查问题，通常资源瓶颈包含以下几种。

1) 后端服务能力不足

集群压力测试的压力在整个集群上，包括后端服务的压力，当服务容量不足时，RT 会明显升高，当超过应用调用服务的超时时间时，应用会出现明显的超时异常。所以要确定这个问题，要看应用依赖服务的耗时是否有增加。

2) 本机内存、CPU 访问瓶颈

在 Java 构建的系统中，内存出现瓶颈，很明显 GC 的次数会增加，这个问题很容易发现，GC 次数变多，相应 QPS 会很低，同时 CPU 的消耗也会明显变大，RT 也会飙升。出现 CPU 瓶颈的标志是 Load 超过应用所在虚拟机配置的 CPU 核数、任务排队，同样 RT 也会飙升。注意，集群压力测试能够发现这种问题，单机引流压力测试同样也能够发现这种问题，所以这不是机器水平扩展的瓶颈。

3) 物理机网卡限制

宿主机的网卡一般都是双网卡配置，双网卡的流量上限一般为 2GB，而定位这个问题需要查看物理机的网卡监控。另外，如果开启 IPtables（防火墙），需要查看 `net.IPv4.netfilter`。



IP_conntrack_max 的配置，因为这个参数设置较小会导致网络丢包，进而造成 TCP 重传，最终导致耗时大量增加。

注：IP_conntrack_max 是允许的最大跟踪连接条目数，Linux 2.4 以下版本的查看命令为 `cat /proc/sys/net/IPv4/IP_conntrack_max`，Linux 2.6 以上版本的查看命令为 `cat /proc/sys/net/IPv4/netfilter/IP_conntrack_max`（old /proc/sys/net/IPv4/IP_conntrack_max is then deprecated!），查看当前的 IP_conntrack_count 的命令为 `cat /proc/sys/net/IPv4/netfilter/IP_conntrack_count`。

4) 交换机网络瓶颈

大型网站机房的机器部署都在机柜或者机柜所在的机框里。对于机架式服务器，1U 的机柜可以放置 24 台物理机，每台物理机的网卡平均上限为 20GB（2 条线）/24 = 0.8GB。对于刀片式机柜，每个机柜包含 4 个机框，每个机框可以放置 16 台物理机，如果是千兆网卡，平均上限为 2GB（两条线）/64 = 0.32GB，如果是 1 虚 4，每台允许的流量将更小，这就是网络出问题的原因。

5) 数据库瓶颈

数据库常见瓶颈有两个，第一个是连接数瓶颈，例如之前交易都依赖于 Oracle 数据库，无法水平扩展，Oracle 连接数上限为 2000，而应用都通过二方库直连数据库，应用的 QPS 一般偏低（和页面的渲染导致 CPU 消耗过大有关），所以应用的机器数量远大于服务所使用机器的数量，应用部署的分散性导致连接数瓶颈。第二个是数据库本身的 QPS 服务能力相对于集群压力测试的流量压力偏小，例如针对数据库能够承受的峰值 QPS，数据库管理员找到方法对 TOP10 的 SQL 进行压力测试，在预设指标没有实现时，测算出数据库的峰值处理能力，数据库的容量评估通过这种方式有了数据依据，之前只能通过人工确认。

6) 存储瓶颈

实际上对于应用集群压力测试本身而言，存储瓶颈比较少见，因为图片存储大部分都从 CDN 走了，所以 CDN 链路上会出现这个问题，但是集群压力测试是通过回放访问日志来实现的，而这些日志不会访问图片或者资源。这也是集群压力测试不能覆盖的一个点，希望大家关注 CDN 的容量，特别是源站的容量问题。

4. 实战过程全景重现分析

在本案例中，对集群进行压力测试时发现集群中某些机器的 RT 表现有很大的不同，有些机器的 RT 特别大，大促技术保障目标是 4 万 QPS，在开始集群压力测试时集群极限能够提供的 QPS 只能达到 11000 多。

第一次压力测试结果如图 11-7 所示。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

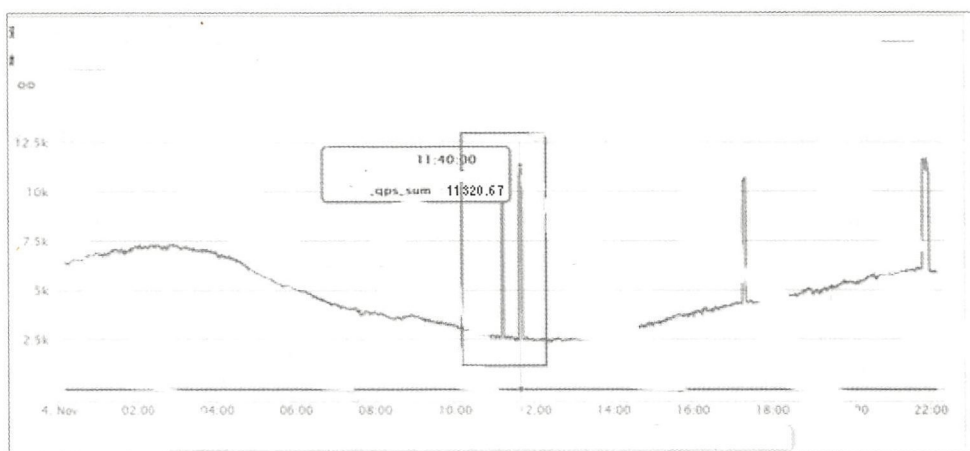


图 11-7

根据以往的经验，很自然地逐一查看应用的数十台机器，看看哪些机器的 RT 飙升了，结果集群中有部分机器 RT 的飙升是其他机器的 4~5 倍，如图 11-8 所示（看趋势线）。

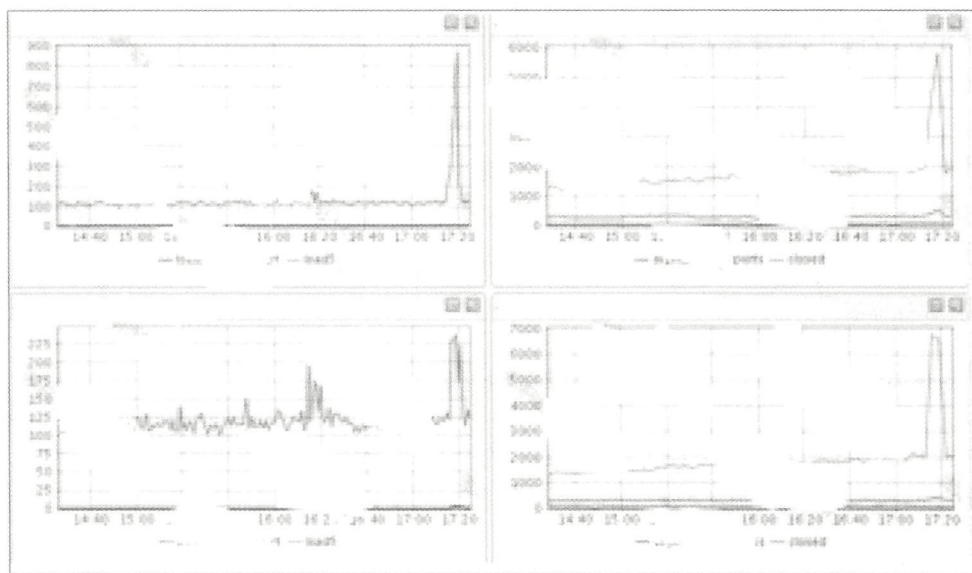


图 11-8

如图 11-8 所示，当时只能达到 11000 QPS，正常的机器耗时 120ms 左右，非正常的机器出现 200ms 到 2s 的耗时。部分机器出现问题，很可能是因为网络有差异（不同的机器在不同的机柜上）。首先定位时发现，所用机器都是两核配置，出现问题的机器配置在两台宿主机上，这



是导致问题出现的主要原因之一。当然 CPU 核数只是原因之一，因为当时出现问题的机器 CPU 的 Load 是 4 到 5，即使是两核配置，RT 的延时超长，不会达到秒级以上，CPU 的任务队列排队不会引起非常大的延迟。所以问题的定位是一个逐步的过程，一个问题解决后，随着水位的提升，木桶的新短板又会出现。先解决已经知晓的问题，对 QPS 的提升会有明显的效果。

定位到 CPU 的核数配置过低的问题之后，将配置进行升级，弃用原来的机器，在别的机框中重新部署该应用，重新进行压力测试，QPS 超过 15000，如图 11-9 所示。其实换机框重新部署应用有另外一个隐形的收益，即网络的问题得到了缓解。

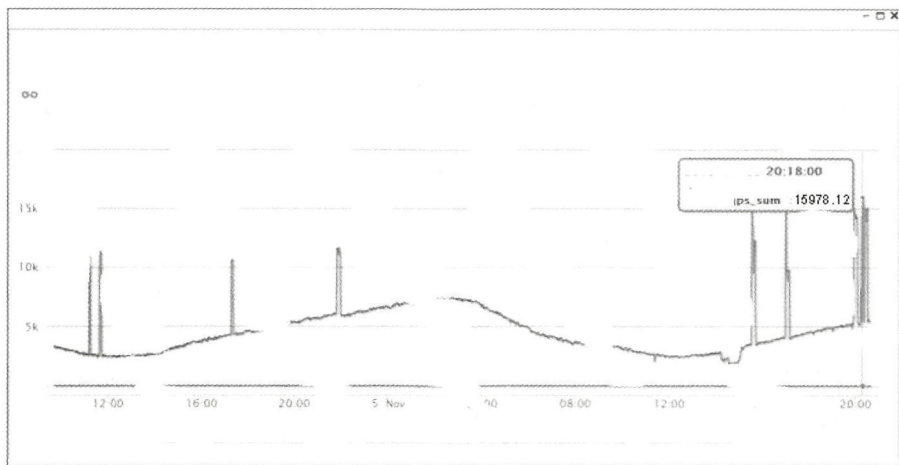


图 11-9

之后对出问题的机柜进行了万兆网卡升级，QPS 达到近两万，如图 11-10 所示。

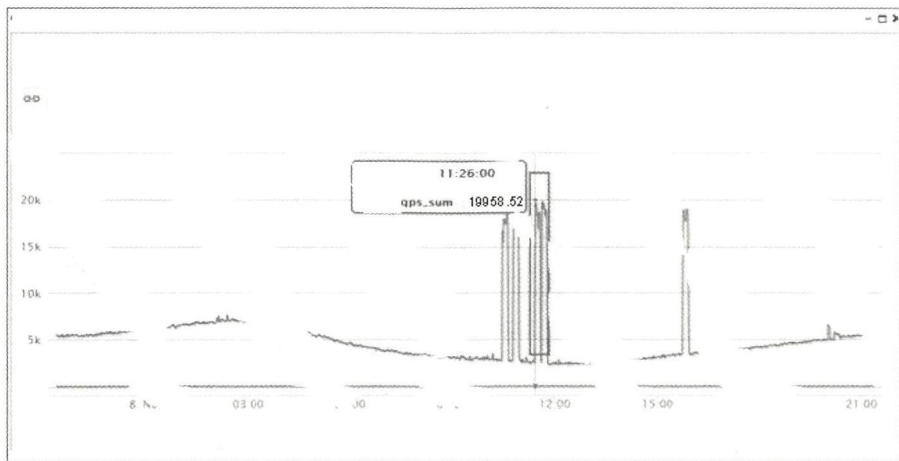


图 11-10



在之后进行的压力测试中，得到一个令人非常沮丧的结果，QPS 又回到了不到 10000，很快定位到原因，是因为在进行清理磁盘，磁盘相关的读写可以查看 IOWait 的情况，如图 11-11 所示。

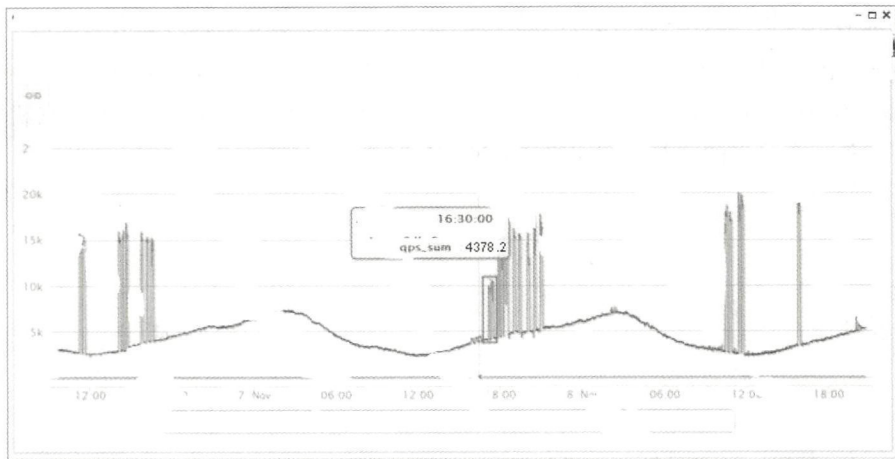


图 11-11

网络瓶颈其实是造成 QPS 低的主要原因，CPU 的任务排队，如果是 Load 4，理论上的 RT 应该最多是正常的两倍，不会出现秒级以上的 RT。之后继续压力测试时发现很多出现 RT 超长的机器都在 A 机柜内，紧急升级 A 机柜上联交换机的带宽，QPS 提升 2000 多，网络瓶颈的标志是交换机网卡流量达到上限导致丢包，网络延迟增大，一般达到 80% 的网卡流量上限时，丢包开始非常严重，如图 11-12 所示。

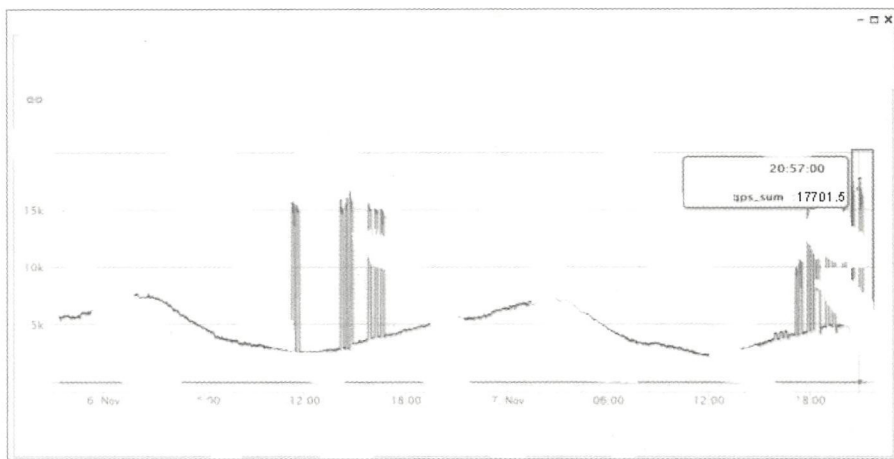


图 11-12



11.7.2 集群个体异常造成的容量问题分析

1. 问题摘要

在上一个案例中，由于机柜上联交换机的带宽容量不足，全链路压力测试达到 20 000 QPS 之后，集群扩充机器，再往上压 QPS 也上不去。针对这个问题，笔者用差异化对比法找到瓶颈机器，将其踢出集群，消除了瓶颈，QPS 从 20 000 增加到 23 300。这个简单的操作为什么能使压力测试 QPS 突然上去了，本节会阐述整个分析和优化的过程。

2. 问题过程描述

上一次全链路压力测试解决了网络问题和机器配置文件引起的 QPS 瓶颈问题，采用了差异化对比法定位瓶颈。但是又出现了新的瓶颈，QPS 达到 20 000 后怎么也上不去了，如图 11-13 所示。

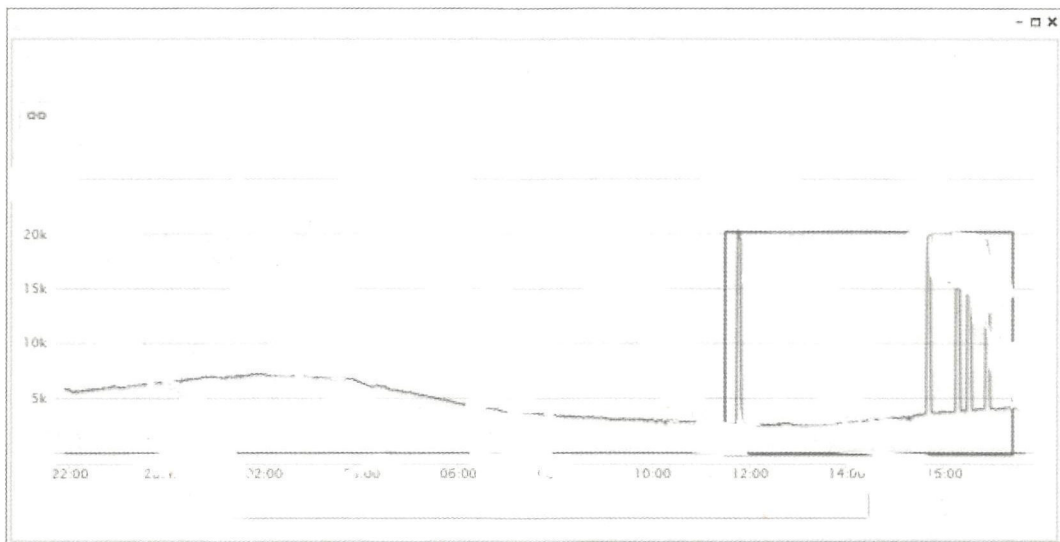


图 11-13

压力测试路径如图 11-14 所示，压力测试目标是代理应用 1，后面的应用 A、应用 B 和应用 C 通过代理应用 1 来代理访问后面的应用。

3. 问题分析

1) 后端应用瓶颈排查

代理应用 1 的 QPS 上不去，首先想到后面的应用是否是瓶颈，查看应用 A、应用 B 和应用 C 的 QPS 和资源的使用情况。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

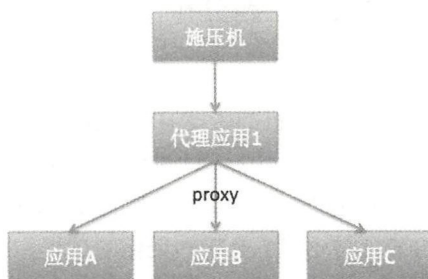


图 11-14

从图 11-15 中可以看出应用 A 的 QPS 很低，远没有达到瓶颈，资源明显未被充分使用。基本可以定位是流量没有达到应用 A 等后面的应用中。

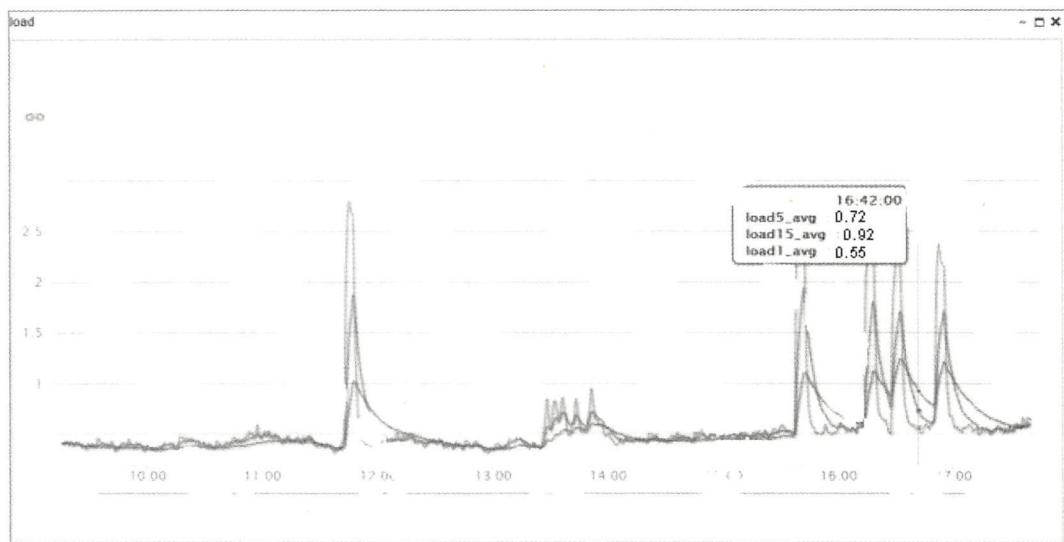


图 11-15

2) 流量上不去的原因分析

首先会想到是否是压力测试机本身的问题，压力测试机的 10 台机器全部是物理机，每台 200 个并发，压力和网卡应该不是瓶颈。其次会想到是不是代理应用 1 本身的 proxy 存在问题，阻挡了后面集群的压力。通过集群中机器的差异表现对比，将代理应用 1 中存在 RT 超长的机器找出来，很快就找到了两台机器，其 RT 达到 3s，如图 11-16 所示。



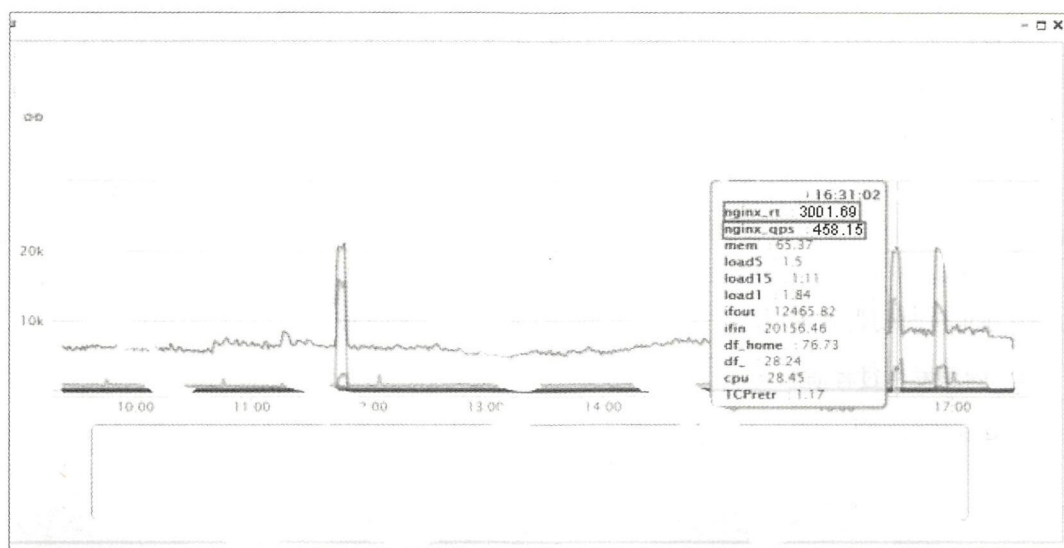


图 11-16

3) 两台机器存在问题会不会是集群瓶颈的原因

在分析之前,先尝试把有问题的机器从集群中去除,再次压力测试,结果如图 11-17 所示。

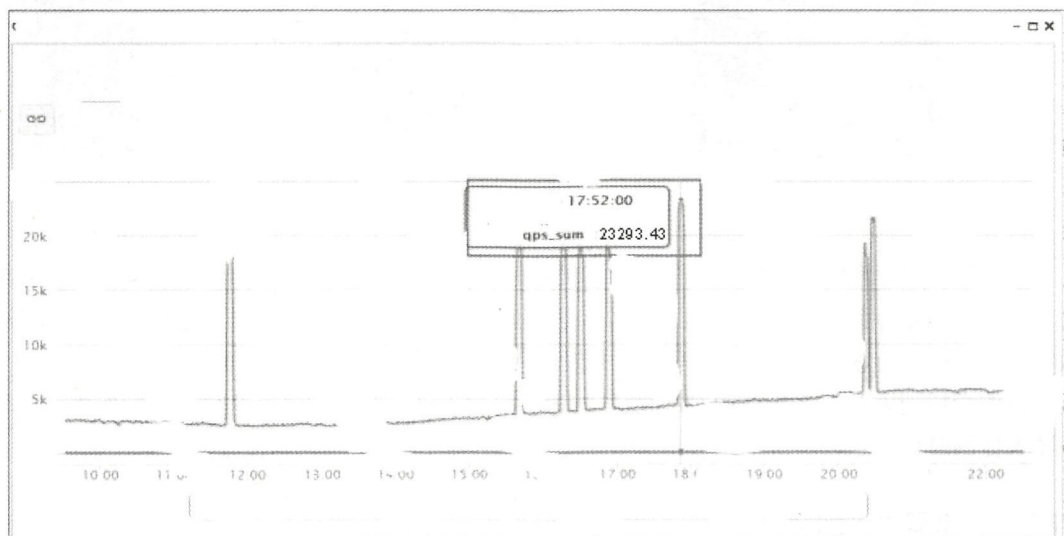


图 11-17

奇迹出现了，QPS 达到 23 000 多了，一下提高了 3000 多，在没有加机器的情况下，整个集群的吞吐能力有明显的上升。

4) 再度分析流量上不去的原因

还是由果到因吧，再看看那台有问题的机器的 QPS 和 RT：QPS = 458，RT = 3s。要达到 458 的 QPS，需要的并发数为 $458 \times 3 = 1374$ 。总共压力测试 2000 个并发，这个有问题的机器占用了近 1400 个并发。注意它的 RT 是 3s，经过它 proxy 到后端的流量速度非常慢，到达后端的并发在单位时间内骤然减速，这就是导致后端流量上不去的原因。

4. 问题发生过程演示

问题发生过程演示如图 11-18 所示。

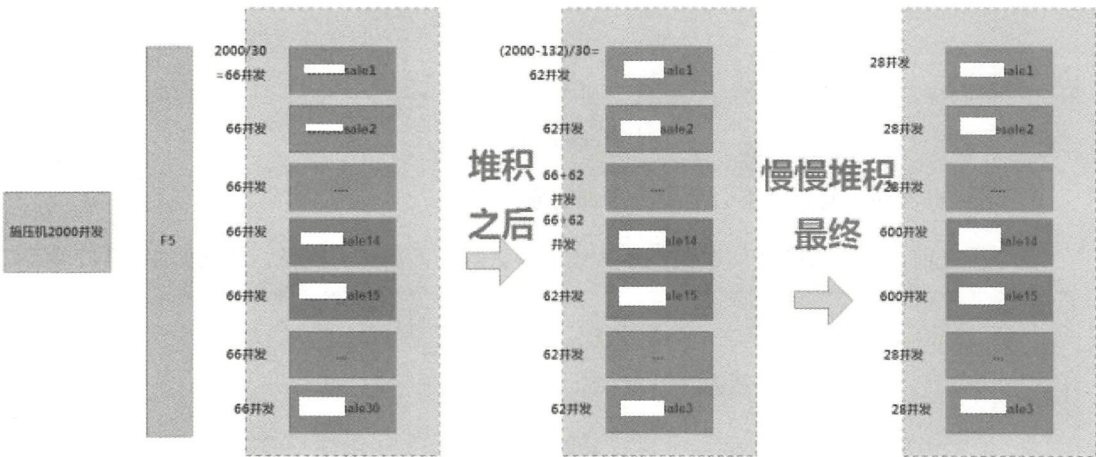


图 11-18

4 层负载均衡器均分并发，当存在黑洞机器时，请求并发慢慢堆积，直到最终堆积越来越大，后续的流量分到其他机器越来越多。这个问题的发生证明，在请求一定的情况下，黑洞确实是存在的。当然在实际中，用户突然并发时，流量对于负载均衡器后面的机器来说并不大，但是请求量特别大时整个集群的 QPS 服务能力会急剧下降，要尽量避免这种情况的发生，尤其是在大促的时候，这种情况下扩容也解决不了问题。

11.7.3 诡异的网络瓶颈

1. 问题现象描述

一般通过容量的线上压力测试来验证集群提供的 QPS 是否和容量评估的目标相符，如果高

于容量评估的目标，那么容量保障达标，反之如果没有达到，继续找瓶颈，直到找到并消除瓶颈为止。在一次大促保障过程中，遇到了非常棘手的问题，某个线上业务的 RT 在压力测试期间抖动非常大，如图 11-19 所示。

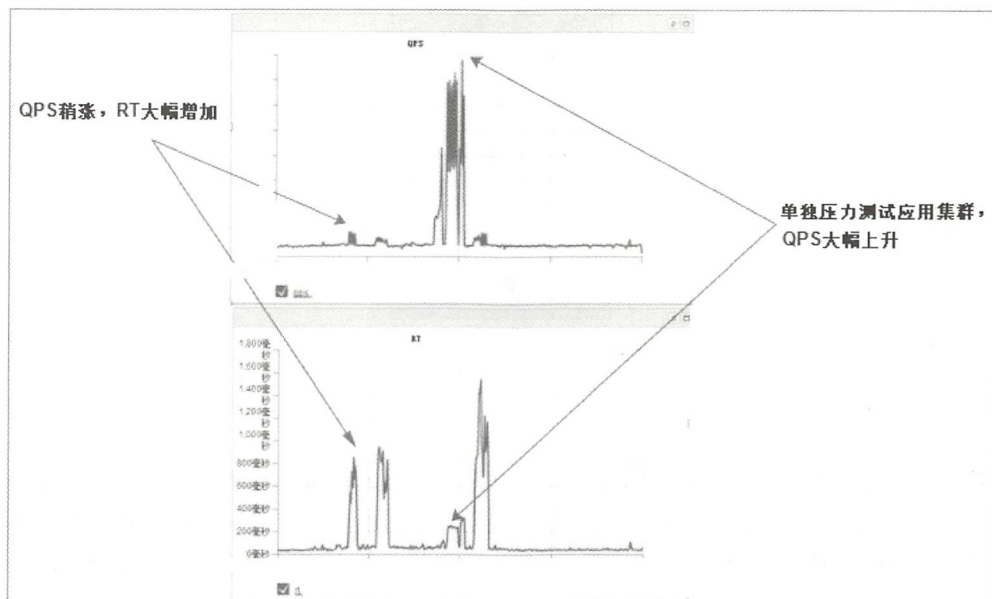


图 11-19

从图 11-19 可以看到，压力测试期间，此应用 QPS 增幅很小，RT 大幅增加，说明集群存在瓶颈，但是压力测试的流量非常小，流量如此小，为什么会这么快达到瓶颈？

于是为了排除应用本身的问题，停止全链路压力测试，对应用集群进行单独压力测试，发现 QPS 大幅增加，RT 抖动变小，说明应用本身没有瓶颈，那么有没有可能是某些公共服务、组件或硬件设施导致的？

2. 问题深入分析

分析这个问题之前，先看一下相关的背景。

1) 应用交互

针对这个应用的交互过程（图 11-20）进行逻辑分析，查找 RT 抖动来源。

从图 11-20 看，A 应用的调用非常简单，只是从 B 服务获取数据返回给 A 应用，再由 A 应用组装数据返回给客户端。

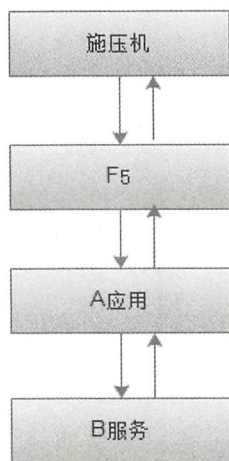


图 11-20

2) 初步分析

集群压力测试时应用正常，说明是压力测试期间某些公共服务、组件、网络等资源过大消耗影响了 A 应用。从图 11-20 可以看出，应用调用十分简单，A 应用只依赖于 B 服务，B 服务是独占的，只有 A 应用才会调用，B 服务不是问题。从服务上可以排除，下面再深入研究一下 A 应用和其他应用的相互影响。

3) 深入分析

从 TCP 层深入分析应用间的交互，如图 11-21 所示。

第 1 步，请求包从施压机发出，通过上联交换机转发给核心交换机。

第 2 步，核心交换机读取包头中的 MAC 地址，根据目标 MAC 地址和端口关系映射表（第一次映射关系不存在，需要广播）找到数据包，将其复制到 F5 连接的端口，转发给 F5 的 4 层负载均衡器。

第 3 步，F5 接收请求包，通过负载均衡策略，确定请求的 Real Server（真实服务器），将原数据包中的目标地址进行修改并重新封装，NAT 到这台 Real Server，经过重新封装后的数据包，同样要转发给核心交换机的某个端口。

第 4 步，核心交换机根据目标 Real Server 的 MAC 地址和端口关系映射表，复制数据包，将其连接到机柜上联交换机的端口，并转发给连接的交换机。

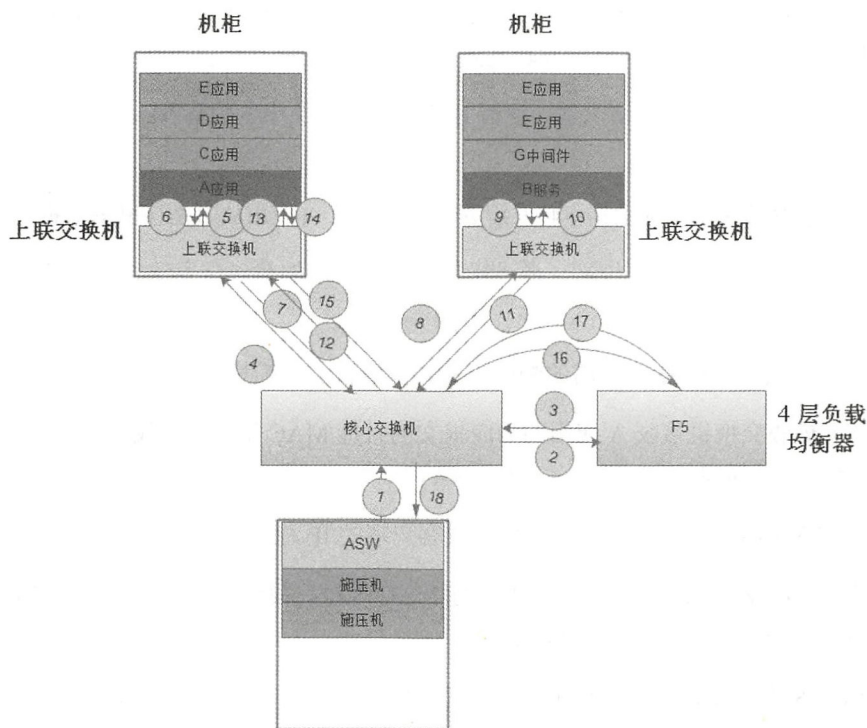


图 11-21

第 5 步，上联交换机收到请求包，解析包头中的目标 MAC 地址，从映射表中找到端口，并转发数据包给 Real Server。

第 6 步，Real Server 接收请求包并解析请求，在本地进行逻辑处理并发起到 B 服务的请求，对 B 服务的请求包进行解析。

第 7 步，A 应用将 B 服务的请求包发送给核心交换机的某个端口。

第 8 步，核心交换机解析包头中的 MAC 地址，根据端口和 MAC 地址找到转发的端口，并将数据包复制到这个端口，转发到对应的机柜上联交换机。

第 9 步，上联交换机收到请求包，解析包头中的目标 MAC 地址，并从映射表中找到端口，将数据包转发给 B 服务。

第 10 步，B 服务接收请求包，解析请求，在本地进行逻辑处理并响应，响应包原路返回给上联交换机。

第 11 步，核心交换机解析 B 服务响应包头中的 MAC 地址，根据端口和 MAC 地址的映射关系找到对应的端口，将数据包复制并转发到对应的机柜上联交换机。

第 12 步，上联交换机收到请求包，解析 B 服务响应包头中的目标 MAC 地址，并从映射表中找到端口。

第 13 步，将 B 服务的响应数据包转发给 A 应用，此时 A 应用从 B 服务获取数据的过程完毕。

第 14 步，A 应用组装响应报文，通过操作系统、网卡的封包转发给 A 应用所在机柜的上联交换机。

第 15 步，上联交换机将 A 应用的响应报文转发给核心交换机。

第 16 步，核心交换机接收 A 应用的响应报文的目标 MAC 地址，并找到对应的端口，复制响应报文到此端口并转发给 F5 负载均衡器（NAT）。

第 17 步，F5 再将响应包中的目标地址改成客户端的 IP 地址重新封包，转发给核心交换机。

第 18 步，核心交换机解析响应包头的目标 MAC 地址，找到转发的端口，并将数据复制到此端口，然后转发给端口连接的上联交换机。

经过上面 18 个步骤，同一个机柜其他的应用同时进行压力测试，从 B 服务的服务器端监控来看，RT 抖动不大，唯一共享的资源是网络资源，单集群压力测试没有出现问题，说明很可能是网络资源引起丢包导致了网络延迟增加。

上联交换机的带宽上限是千兆，每个机柜有 16 台物理机千兆网卡，物理机网络的收敛比是 1:16，一台物理机虚拟 4 台服务器，网络收敛比为 1:64。查看物理机丢包，F5 负载均衡器均未出现丢包，上联交换机的网络流量大概在 600MB，均未达到上限，是不是网络有问题？陷入了僵局。

经过网络工程师的排查，发现从各个层面都看不到明显的丢包，问题还在继续，一旦压力测试，A 应用的 RT 会有很大变化。因为应用部署较为简单，于是把 A 应用全部打散重新部署，问题一样出现，RT 抖动没有消失。

3. 问题转机

经过几次调整均没有变化，是否需要把 B 服务的数十台服务器全部重新部署，B 服务涉及数据的重新构建，迁移成本非常高，如果把 B 服务所在机柜的其他应用全部移走，代价也非常大。确实陷入了僵局，大促也迫在眉睫。

但是问题的排查还要坚持下去，经过进一步查看，在 B 服务的机框的服务器列表中发现了一个可疑点，B 服务所在的机框和 Tair 部署在一起，从下面的机框部署可以看出，B 服务和超大流量的中间件 Tair 放在了一起。

1) 一台 B 服务机器所在的机框机器列表：

```
3-DC.NET d01.hst 3-DC.NET d02.hst 3-DC.NET a01.hst 3-DC.NET a02.hst 3-DC.NET a03.hst
3-DC.NET a04.hst 3-DC.NET a05.hst 3-DC.NET d01.hst 3-DC.NET d02.hst 3-DC.NET a01.hst
3-DC.NET a02.hst 3-DC.NET a03.hst 3-DC.NET a04.hst 3-DC.NET a05.hst 3-DC.NET tair02.hst
3-DC.NET c01.hst 3-DC.NET c02.hst 3-DC.NET c03.hst 3-DC.NET c04.hst 3-DC.NET c05.hst
3-DC.NET B2.hst
```

2) 另外两台 B 服务机器所在的机框机器列表：

```
2-DC.NET g01.hst 2-DC.NET g02.hst 2-DC.NET g03.hst 2-DC.NET tair01.hst 2-DC.NET
e01.hst 2-DC.NET e02.hst 2-DC.NET e03.hst 2-DC.NET f01.hst 2-DC.NET f01.hst 2-DC.NET
f01.hst 2-DC.NET B1.hst 2-DC.NET B3.hst
```

于是把这 3 台 B 服务的机器移到流量小的机框里，再进行压力测试，A 应用的 RT 表现十分平稳，问题解决。

4. 总结

解决这个问题实际上花了两周时间，过程中做过很多尝试，也曾陷入僵局，特别是在 A 应用换机框时一点都没有改进，对 B 服务进行移机框操作比较复杂，如果 B 服务有多台机器，不能把所有的机器全部换机框，最后排查出来只有一台机器有问题。在网络数据没有达到上限的情况下，没办法证明是网络的原因，只能通过逻辑推理定位到问题机框。还是那句话，办法总是比问题多，在排查问题陷入僵局时，需要用逻辑推理和快速测试验证的方法进行定位和解决。

11.7.4 多机房压力测试流量不均问题分析

1. 问题概述

在某次 618 大促保障过程中，遇到了非常多的问题。首次启用了双机房共同保障策略，由同城双机房共同向外提供服务。由于多种原因，实际上双机房并不是真正的双 A 架构，而是通过智能 DNS 按比例分流给两个机房，由于老机房的机器无法扩容，网络也无法升级，所以导致新机房会承担更多流量，在保障过程中，遇到最多的是流量不均的问题，不能确定这个问题是否跟智能 DNS 的解析相关。智能 DNS 按比例分流的算法采用最简单的负载均衡算法 roundrobin



或者 roundrobin-weight，本节将对大促全链路压力测试过程中的“流量不均问题”进行剖析。可以肯定的是，所谓的流量不均是大促大家的误解，下面希望找一些证据来分析导致误解的原因。

2. 当前双机房架构简析

某项业务有两个机房，两个机房通过智能 DNS 按照一定比例分流的方式将用户流量导入新老机房（类似于 CDN 的全局调度器对于一个 region 的分流策略），然后由新老机房同时对外提供服务，对外来说是一个逻辑机房。

在没有全局调度之前，Local DNS 把解析请求给权威 DNS，权威 DNS 直接返回给 Local DNS A 地址。在有全局调度之后，权威 DNS 将解析请求给全局调度器，由全局调度器 DNS 直接返回给 Local DNS A 地址，如图 11-22 所示。

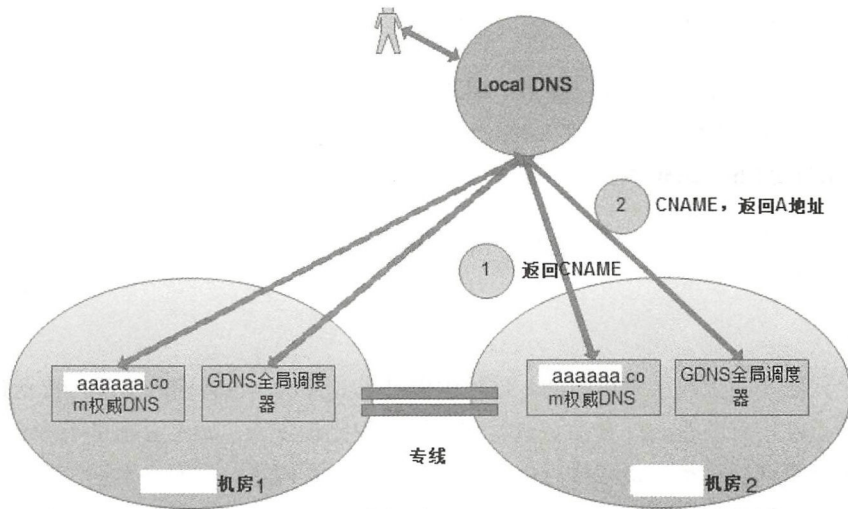


图 11-22

全局调度器轮询策略分为两种：对于流量比例 10：10，返回策略有 ABABABABAB 和 AAAAABBBBB 两种流量分配方案，对于请求量大的域名这两个策略无影响，对于请求量小的域名第二种策略就显得不太均衡了，当 TTL 较长时就会有波浪式流量。考虑到性能，选择第二种策略居多。

3. 问题的发生

在大促保障过程中，发现机房 1 和机房 2 的压力不均衡，某些应用在一个机房由于压力过载导致 CPU 消耗很高，而另外一个机房的资源过度闲置，机器无法充分利用。



如图 11-23 和图 11-24 所示，www.aaaaaa.com 域名是按照 1:1 的流量比例对智能 DNS 进行分流的，但是在压力测试的时候，从 QPS 统计数据来看，两个机房的流量十分不均衡，比如在 19:25，机房 1 的 QPS 是 20 000 左右，机房 2 的 QPS 是 656，相差 30 多倍，在另外一个时间点，如图 11-25 和图 11-26 所示，相差 3 倍多，是智能 DNS 分流的算法出现问题了吗？

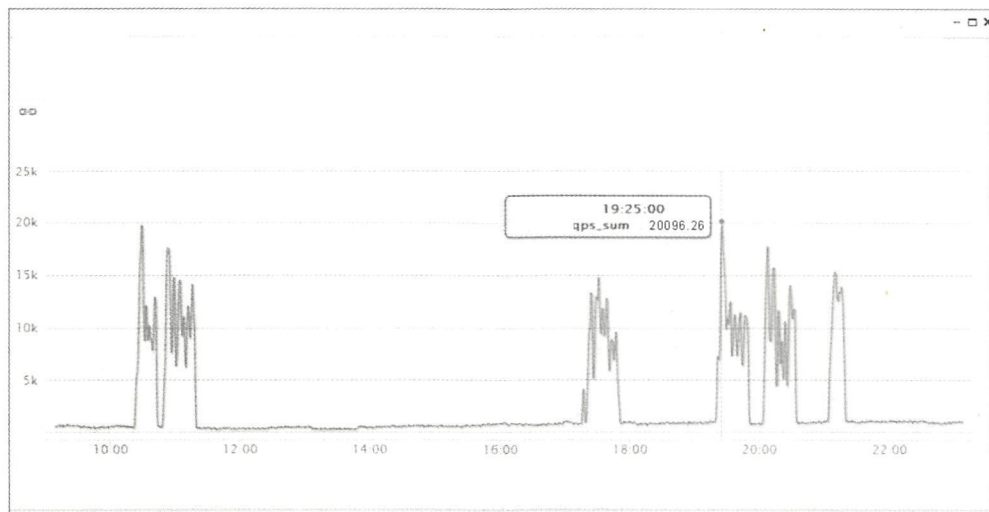


图 11-23

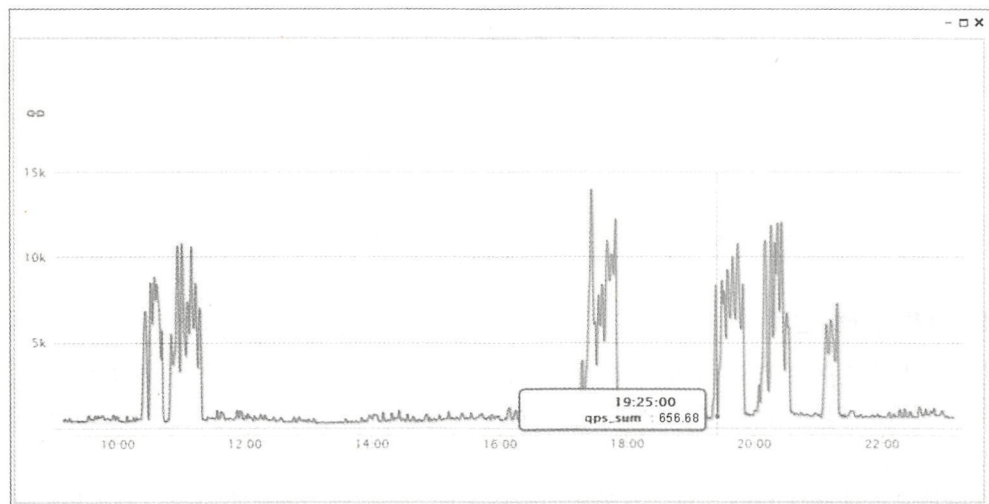


图 11-24



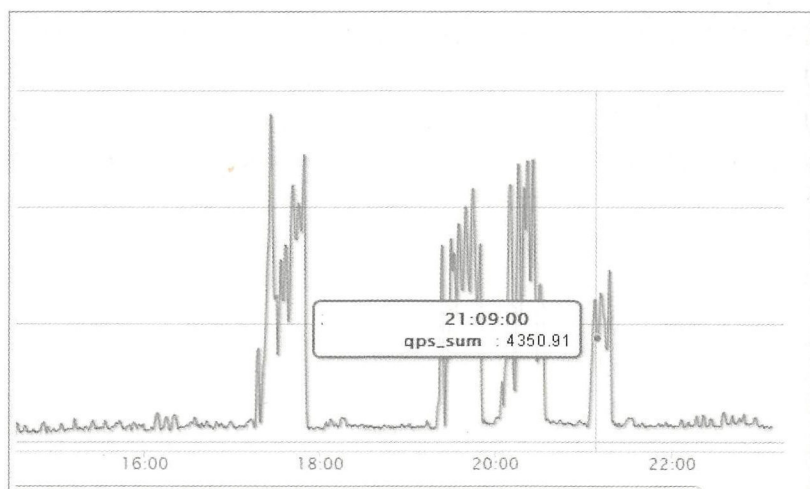


图 11-25

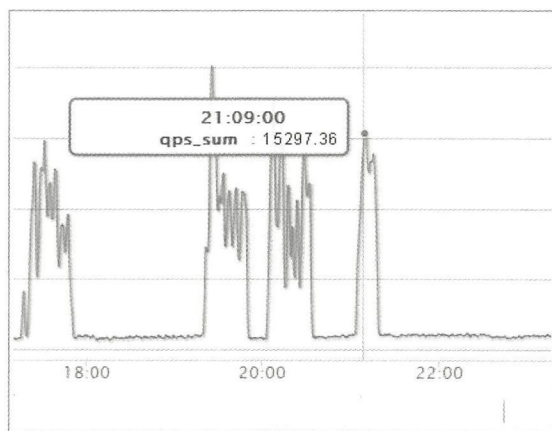


图 11-26

4. 问题的分析过程

在确定是否是智能 DNS 问题之前，我们想一想智能 DNS 是干什么的。智能 DNS 只做了一件事情，那就是把域名解析成 IP 地址，其分配方式是轮询，轮询是指每发起一次 DNS 查询请求，就交叉分配一次 IP 地址，理论上这个分配肯定是均衡的。在集群保障过程中，要测量和验证系统能够承受的峰值 QPS。对于双机房的架构来说，如果压力测试上不去，除了集群存在某些瓶颈（如网络瓶颈、引擎瓶颈、中间件瓶颈等），QPS 会表现不均衡，给保障带来很大难度，一个机房的压力很大，而另外一个机房的资源闲置，造成资源的浪费。



就上述问题来分析一下原因。www.aaaaaa.com 的域名查询请求过程如图 11-27 所示。

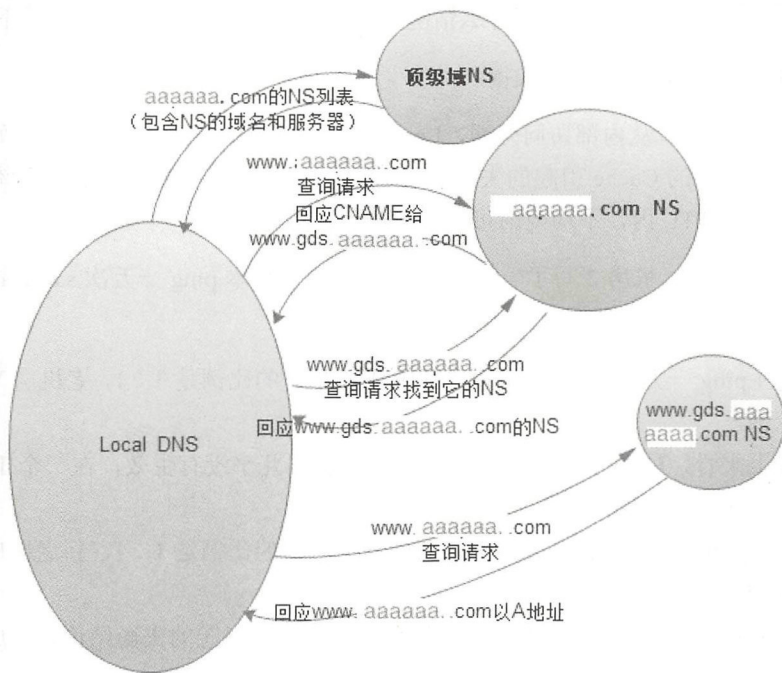


图 11-27

DNS 解析分为 3 步。

第 1 步, 在第一次访问时, 浏览器和操作系统及 Local DNS 没有任何缓存, Local DNS 将 DNS 查询请求给 aaaaaa.com 的权威 DNS 服务器, 权威 DNS 服务器将 CNAME 指令返回给 Local DNS, 并指向 www.gds.aaaaaa.com 的智能 DNS 服务器。

第 2 步, 向 aaaaaa.com 权威 NS 寻找 www.gds.aaaaaa.com 的 NS, 权威 NS 返回给 Local DNS www.gds.aaaaaa.com 的 NS 列表, 包含域名和 IP 地址。

第 3 步, Local DNS 会根据一定的规则, 例如就近选择一个 NS, 将 www.aaaaaa.com 的域名解析请求发送给 www.gds.aaaaaa.com 的一个 NS, 这个 NS 是智能 DNS, 可以根据 RoundRobin 的比例, 分配给请求者。

从上面的原理可以看出, 域名解析请求在长时间内是均衡的, 是严格按照比例来进行解析的, 例如 www.aaaaaa.com 的新老机房的外网 VIP, 配置成 1:1 的关系, 那么从整体的解析请



求次数来看，总体肯定是 1 : 1 的关系。

什么会影响 QPS 的比例关系？即什么情况下会发生在 1s 内请求和流量出现不均衡？

1) Local DNS 的 Cache 会引起失衡

关于这种情况，笔者从内部访问外网，Local DNS 在自己的机房内部。笔者做了一个实验，可以判断是否是 DNS 的 Cache 引起的失衡，可以确定的是，Local DNS 及操作系统本身有 DNS Cache，那么在一个 TTL 内，不均衡的可能性很大。

笔者分别在机房 1、机房 2 写了一个 ping 脚本，这个脚本 ping 一万次 s.aaaaaa.com 域名，从 ping 的情况来看得出以下数据。

- 在机房 1 ping 一万次 s.aaaaaa.com，其中新老 VIP 的比例是 3 : 1，老机房 VIP 获得了更多的轮询。
- 从结果上来看，Local DNS 的 Cache 和 OS Cache 几乎没有生效，在一个 TTL 内，两者都得到了轮询命中的机会。
- 在机房 2 ping 一万次 s.aaaaaa.com，其中新老机房的比例是 1 : 1，新老机房得到机会均等的轮询命中。

从上面的数据来看，从机房施压，Local DNS 的 Cache 引起的失衡结论是不成立的，因为 TTL 并没有生效。

2) 网络包到达速度引起的流量不均衡

一秒之内处理完成的请求数量很关键，在压力测试过程中，发现了一个现象，QPS 曲线会出现陡然下降的情况。如图 11-28 和图 11-29 所示，在 18:30 分时，QPS 下降，对应的 RT 也下降， $QPS = \text{并发数} / RT$ ，QPS 下降的原因只有两个，一个是压力过载，另一个是压力变小。目前双机房峰值 $QPS = \text{老机房峰值} QPS + \text{新机房峰值} QPS$ （在同一时刻），当 QPS 下降、RT 也下降时，只能说明压力测试的流量在短时间内是不稳定的，当新老机房在一秒内的流量达到速度不一样时，会出现一个机房的 QPS 很高，另一个很低，两个之和会远小于两个的峰值。更可怕的情况是，一个机房过载，一个机房的资源闲置，导致资源无法最大化利用。

3) 是否真的不均衡

仍然要强调一下一秒之内处理完成的请求数量，如上面的 DNS 解析原理中提到的，智能 DNS 只做了一件事情，就是按比例给用户分配新老机房的 VIP，算法是非常简单的。对于一秒之内的请求完成而言，可能是因为网络包到达速度不一样而导致请求无法及时到达各自被智能 DNS 指定的 VIP 上。



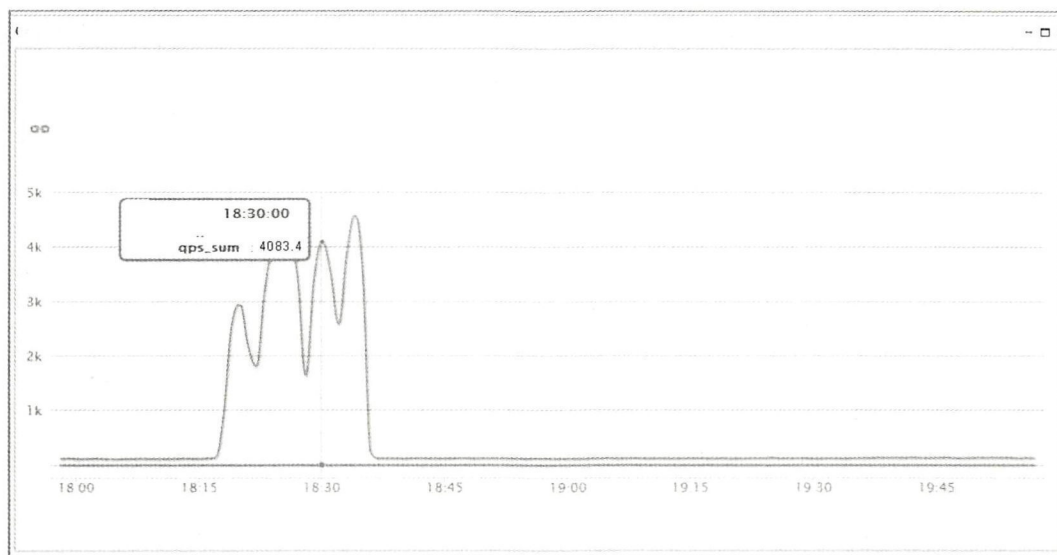


图 11-28

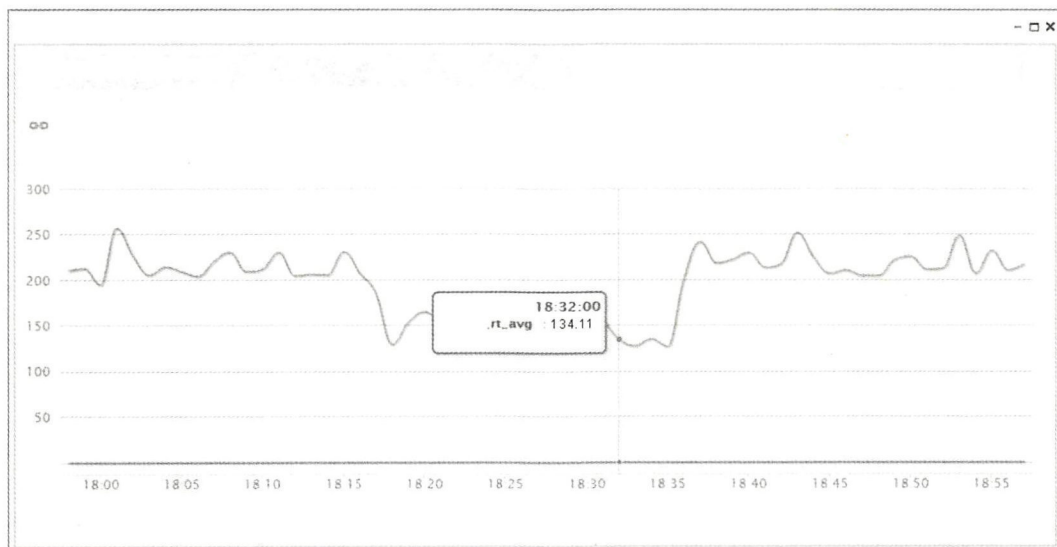


图 11-29

为了证明这个结论，看了一下压力测试 10 分钟之内的请求完成情况，得出以下结论，在 10 分钟之内请求完成的数量完全和 DNS 分配的比例相同，这足以说明 DNS 的比例分配是没有问题的。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

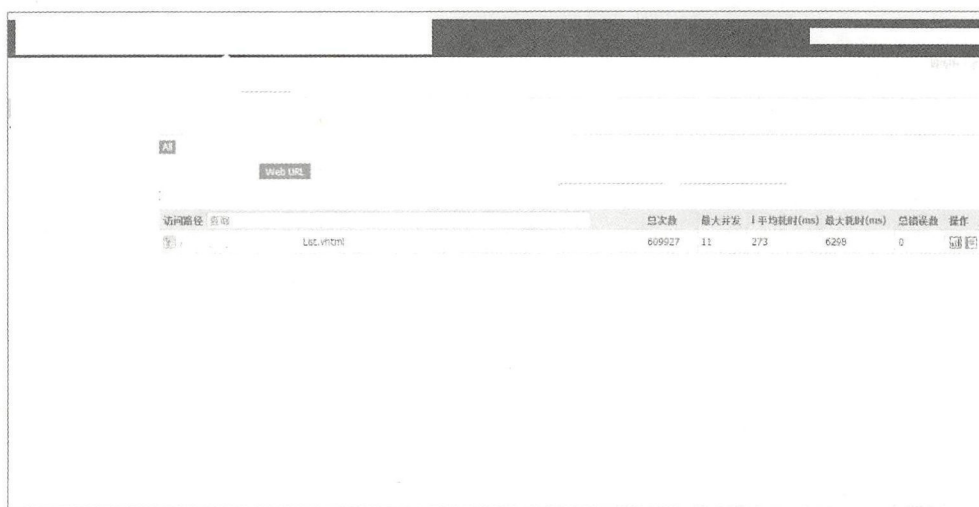
如图 11-30 和图 11-31 所示，分别计算了新机房和老机房的请求访问次数，发现比例和 DNS 指定的 7:3 相符合，查看其他几个应用的情况也是如此。



The screenshot shows a monitoring dashboard with a table of request statistics. The table has columns for '访问路径 | 页码' (Access Path | Page Number), '总次数' (Total Count), '最大并发' (Maximum Concurrent), '1 平均耗时(ms)' (1 Average Response Time (ms)), '最大耗时(ms)' (Maximum Response Time (ms)), '总错误数' (Total Error Count), and '操作' (Action). The data row for 'list.html' shows a total count of 265572, maximum concurrent of 8, average response time of 284 ms, maximum response time of 3267 ms, and total error count of 0.

| 访问路径 页码 | 总次数 | 最大并发 | 1 平均耗时(ms) | 最大耗时(ms) | 总错误数 | 操作 |
|-----------|--------|------|------------|----------|------|---|
| list.html | 265572 | 8 | 284 | 3267 | 0 |   |

图 11-30



The screenshot shows a monitoring dashboard with a table of request statistics. The table has columns for '访问路径 | 页码' (Access Path | Page Number), '总次数' (Total Count), '最大并发' (Maximum Concurrent), '1 平均耗时(ms)' (1 Average Response Time (ms)), '最大耗时(ms)' (Maximum Response Time (ms)), '总错误数' (Total Error Count), and '操作' (Action). The data row for 'list.html' shows a total count of 609927, maximum concurrent of 11, average response time of 273 ms, maximum response time of 6258 ms, and total error count of 0.


| 访问路径 页码 | 总次数 | 最大并发 | 1 平均耗时(ms) | 最大耗时(ms) | 总错误数 | 操作 |
|-----------|--------|------|------------|----------|------|---|
| list.html | 609927 | 11 | 273 | 6258 | 0 |   |

图 11-31

5. 问题结论和方案遐想

综上分析，得出结论。

1) 按比例静态分流的方式在任何情况下 QPS 都不可能达到理论中的均衡



按比例静态分流的方式只能保证总体访问次数和比例一致，不能保证短到 1s 之内的流量均衡。要做到 1s 以内的均衡，必须根据机房的实际压力进行动态分流，对此目前还没有解决方案。DNS 的分流策略相对比较简单，要么添加机器，要么动态调整分流比例，目前网络的到达速率和丢包率是无法确定的。

2) 目前的评估方式是不得已而为之的方法

目前的评估方式是用峰值 QPS 之和来评估两个机房所能承受的峰值 QPS，为什么不按照两者最大 QPS 之和进行评估？这是因为两个机房的公用资源太多，例如引擎、算法服务、中间件全部跨机房混合调用，如果按照最大 QPS 之和算，一旦这些公用资源出现瓶颈，将无法评估在峰值情况下这些公共资源是否存在瓶颈。

3) 真正单元化的异地多活架构是避免 QPS 不均衡的根本保障

真正的单元化是指机房之间访问隔离，在本机房内的访问不会跨机房，多个机房的容量评估可以分开评估，它们之间没有公用资源，比如引擎、网络、中间件相互独立，这时完全可以分开评估。但是真正的单元化很难做到，需要完整的全球化架构支持。实际上总是有各种问题，例如一致性要求高的库存更新问题、公共数据多份存储和访问问题。

4) 更长时间的压力测试方式和仿真环境

在网络速度不恒定的情况下，两个机房的 QPS 比例会失衡，压力测试时间越短，这种不确定性越高，加长压力测试时间，可以缓解这个问题。在实际大促过程中，用户访问网络情况是很难预知的，即使使用 CDN 的节点进行压力测试，也会造成 QPS 不均衡，CDN 节点压力测试的方式只能减少采用目前内部机房流量压力测试的方式带来的内部流量的剧增。要做到比较真实的仿真，或许可以使用 TCP Copy 的方式，TCP Copy 的方式可以模拟丢包，但是不能模拟用户真实的网络速度，RTT 在机房内部回放比用户真实情况要小很多。所以要做到正确的集群压力测试和实际保障结果相同，具有隔离特性的单元化多机房架构是终极解决方案。

11.7.5 Tengine 限流案例

在某次大促准备过程中，压力测试某个系统的 QPS，但是到达 3000 后就变成直线了，RT 在到达峰值后，突然从 40ms 变成 80ms，如图 11-32 所示。CPU、Load 都很低，磁盘表现很正常，JMeter 也没有出错，而且随着时间的推移，RT 一直上升，从表象上看，肯定是被限流了，但是奇怪的是，如果是被限流了，RT 应该还是比较稳定的。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

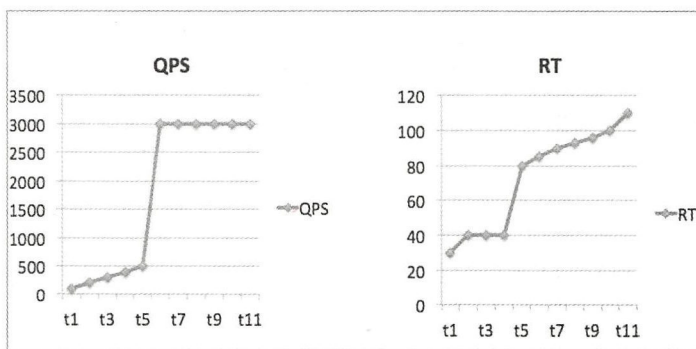


图 11-32

案例分析：QPS 限流配置为什么会有 base_QPS、max_QPS、burst，它们是干什么的？当请求 QPS 超过 base 值时会开始按比例丢掉一部分，一个请求被丢掉的概率和地域权重相关，这些权重在 tmd3_IP.conf 里配置，默认是不需要关心的。当请求 QPS 超出最大值时就开始丢弃超出部分的请求，以上两条的“丢弃”实质上会进入 Burst 的队列排队，Tengine 会 Hold 住这部分请求不处理，等待时间可以理解为当前 Burst 队列里的请求数除以 max_QPS，即最长等待时间就是配置的 Burst/max_QPS，所以在看到 RT 增长时请不要过于惊讶。

总结：连接池类型的瓶颈表象一般都是应用所在服务器的资源消耗小，如果 CPU、Load 等各种关键资源都处于很低的水位，那么非常可能某个连接池存在问题。

11.8 总结

在亲历多次大促之后，现在再去回想，如何才能做一个比较好的大促保障方案，尽量避免大部分问题呢？总结起来，大促主要工作要确保 4 种能力：验证能力、水平扩展伸缩能力、快速发现问题能力、预案能力。

大型网站的大促是一项复杂的系统工程，要有严格的项目管理和进度管理，以及上下游沟通，大促最关键的是提升系统的水平伸缩能力，水平伸缩能力构建的前提是容量的规划、度量及瓶颈问题的发现和分析，最终通过合理的架构模式来提升水平扩展能力。

12

第 12 章

数据分析驱动性能优化

12.1 WebP 性能优化案例背景

前面所有案例都是从技术视角来观察性能问题的，大部分是根据经验来进行性能优化的，这种根据经验进行性能优化的方式是有体系化的结构的，如图 12-1 所示。

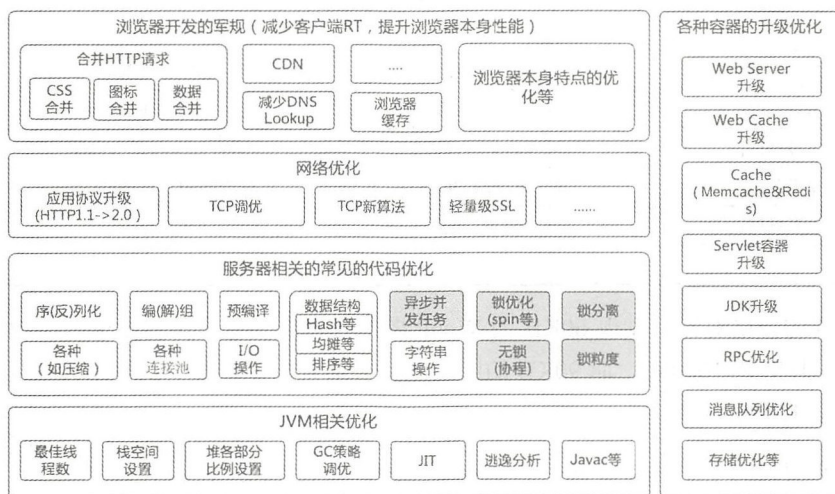


图 12-1

不同的人对这个结构有不同的认识，在遇到问题的时候，就根据请求链路及对链路上每个环节的掌握程度来逐个分析原因。其实这样的方式效率是比较低的，接下来介绍一种更加高效的方式来驱动性能的优化：向性能迭代要数据，向数据要性能迭代。

12.1.1 WebP 格式开始兴起

很多年前发生了这样一件事情，当时 Google 的 WebP 推出了新的图片压缩算法 WebP，很多网站都开始使用 WebP 作为图片的压缩格式。这种压缩算法压缩出来的图片体积更小，但是解压缩时需要占用更多的 CPU 资源，是典型的以时间换空间的算法。

下面举一个真实的例子：

(1) 有一张图片 A，用 JPG 压缩之后的体积是 90KB，用 WebP 压缩之后的体积是 60KB，使用 WebP 之后，体积缩小了 1/3。

(2) 当时的网络 RTT 是 200ms，不要疑惑，确实是 200ms，因为这个 RTT 是跨了大洋的 RTT。

(3) 当时的拥塞窗口初始值是 4，那么第一次请求可以携带的数据量是 $1460 \times 4 = 5840\text{B}$ 。

那么这 1/3 体积的缩小能带来什么助益呢？假设当时的网络情况是理论最优的，没有任何抖动，拥塞窗口的大小为 32，用当时的场景来进行推导，如表 12-1 所示。

表 12-1

| | 90KB的JPG图片 | 60KB的WebP图片 |
|-------------|---------------|---------------|
| 链接时间 | 200ms | 200ms |
| 第1次传输数据量及耗时 | 5840B, 200ms | 5840B, 200ms |
| 第2次传输数据量及耗时 | 11680B, 200ms | 11680B, 200ms |
| 第3次传输数据量及耗时 | 23360B, 200ms | 23360B, 200ms |
| 第4次传输数据量及耗时 | 46720B, 200ms | 19120B, 200ms |
| 第5次传输数据量及耗时 | 2400B, 200ms | 0B, 0ms |
| 总耗时统计 | 1200ms | 1000ms |

通过这样的表格对比，发现 60KB 的 WebP 图片在这个场景下只需要 4 次网络传输，加上链接时间，总耗时是 1000ms，而 90KB 的图片在这个场景下则需要 5 次网络传输，加上链接时间，总耗时是 1200ms。

假设拥塞窗口的平均大小是 16，再来推导一遍，如表 12-2 所示。

表 12-2

| | 90KB的JPG图片 | 60KB的WebP图片 |
|-------------|---------------|---------------|
| 链接时间 | 200ms | 200ms |
| 第1次传输数据量及耗时 | 5840B, 200ms | 5840B, 200ms |
| 第2次传输数据量及耗时 | 11680B, 200ms | 11680B, 200ms |
| 第3次传输数据量及耗时 | 23360B, 200ms | 23360B, 200ms |
| 第4次传输数据量及耗时 | 23360B, 200ms | 19120B, 200ms |
| 第5次传输数据量及耗时 | 23360B, 200ms | 0B, 0ms |
| 第6次传输数据量及耗时 | 2400B, 200ms | 0B, 0ms |
| 总耗时统计 | 1400ms | 1000ms |

可见,在拥塞窗口小的情况下,90KB的JPG图片比60KB的WebP图片多耗时400ms。这已然是一个非常大的数值了。

当然在用WebP之前,当时的性能团队并没有经过这样精确的推导,但还是根据经验决定要用WebP。

12.1.2 WebP改造使L-D转化率下降

通过一番改造之后,WebP上线了,但遗憾的是Apache Bench test中WebP那部分的L-D转化率居然下降了,简直不可思议!于是大家开始找原因,但没有找到,WebP上线的事情就搁浅了。

直到几个月后的一天,笔者无意中发现WebP的TTFB居然高达847ms,而JPG的TTFB只有51ms,如图12-2所示。

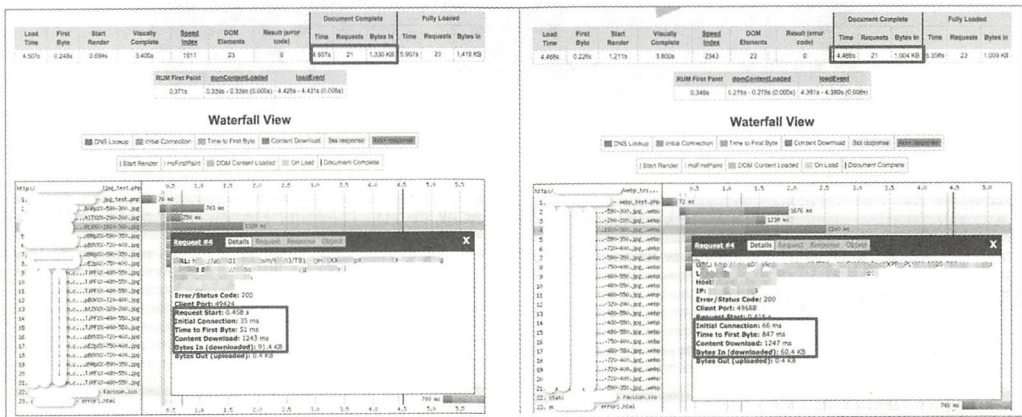


图 12-2

这是什么概念呢？CDN 边缘节点的缓存服务器接收请求之后，加载本地磁盘中的 WebP 图片，然后返回图片数据，第一个返回的字节到达客户端的时间为 847ms，按照计算机当前的发展水平，这是不符合常理的，可以解释的原因只有一个，即那台服务器上并没有这张 WebP 图片。

笔者开始不断地请求这张图片，清了浏览器缓存之后再请求，不管怎么请求，TTFB 始终超过 800ms，这说明并不是恰巧请求到正好没有 WebP 的某台机器上，于是请 Akamai 的接口人确认一下，结果对方的回复是确实没有为 WebP 这种图片类型进行过缓存，一切就说得通了。

这个问题为什么难以判断呢？因为当时 L-D 的影响因素很多，大家无法确定是哪个点影响了 L-D 的转化率，所以当时的场景有些混乱，导致大家没有找到根本原因，就此作罢。

回过头来再看，如果通过数据分析的方式来进行问题诊断，那么可以立即找到问题所在并将其解决。接下来将阐述如何根据数据分析来诊断这个问题。

12.2 性能优化中的数据分析原理与方法

12.2.1 数据分析简介

数据分析简单来说就是使用统计学的相关方法对所收集数据进行分析，以寻找数据中的规律，最后形成结论。数据分析一般会分成如图 12-3 所示的 5 个环节。

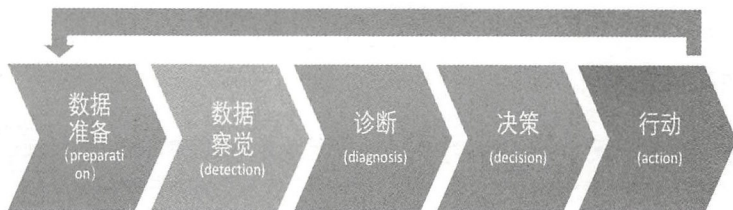


图 12-3

数据分析是一个非常大的领域，很多数据科学家或者 BI 工程师每天都在做数据分析相关的事情，作为技术人员在很多场景下也需要和数据分析打交道，接下来看一下数据分析常规的步骤及其在性能优化中的具体含义。

(1) 数据准备：此处需要明确几个内容。

① 指标定义及指标树。

② 数据采集方法。

③ 数据清洗和存储。

(2) 数据觉察：在数据准备完成之后，可以通过各种分析技巧来查看数据中蕴藏的规律，数据觉察又分成了数据分析技巧和数据分析方法论。

① 分析技巧。

② 分析方法论。

(3) 问题诊断：利用一系列的数据分析技巧找到性能问题的根本原因，对症下药。

(4) 决策并行动。

① 这里的决策是指产出具体的优化方案。

② 这里的行动是指方案的落地实施，同时通过 Apache Bench test 检查效果。

12.2.2 数据分析之杜邦分析

数据分析中常见的分析技巧有对比分析、分组分析、结构分析、平均分析、交叉分析、综合评价分析、杜邦分析、矩阵关联分析等，如图 12-4 所示。

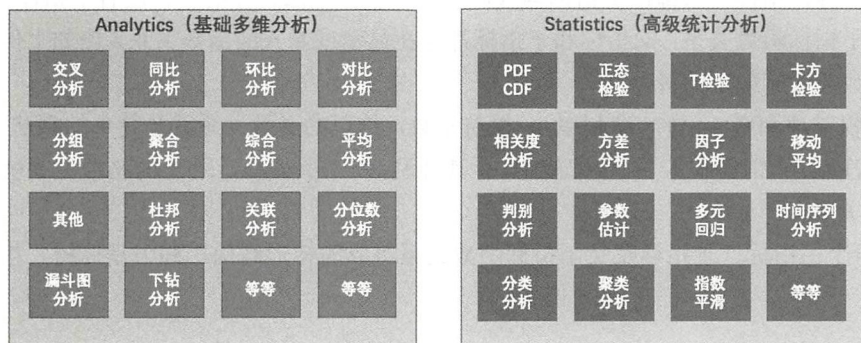


图 12-4

在接下来的案例中会用到杜邦分析的一个变种，所以本节着重介绍杜邦分析。我们先来简单地了解一下什么是杜邦分析，如图 12-5 所示。

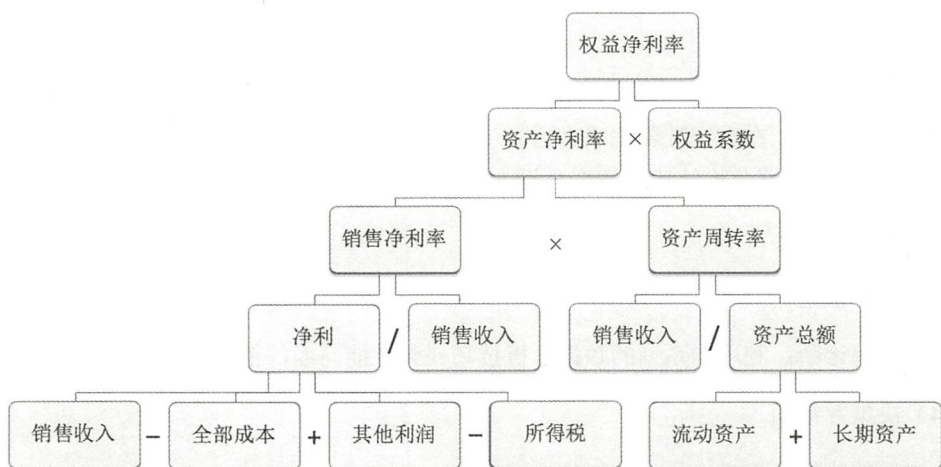


图 12-5

在杜邦分析中，有如下几个术语。

- 指标：衡量目标的单位。
- 数据表：每个指标都来自对数据的统计，所以指标定义完成后必须获得指标所需要的数据表。
- 子指标：指标一般都由子指标构成，当一个指标无法再分解时就没有子指标了。
- 父子指标函数关系：父指标和子指标都存在显示或隐式的函数关系，比如上例中显示的函数关系有如下两种。
 - 权益净利率 = 资产净利率 × 权益系数，这里的函数关系就是简单的乘法关系。
 - 净利 = 销售收入 - 全部成本 + 其他利润 - 所得税，这里的函数关系就是加减法关系。

这样一个由指标、子指标及父子指标函数关系组成的树形结构，称为指标树。指标树有什么作用呢？

指标树有几大好处。

1) 便于绩效管理

明确业务指标及指标分解，便于团队作战（这是管理问题，不是数据分析问题）。

2) 缩小问题范围

(1) 指标向下追溯：当父指标发生波动时，可以自动向下追溯，以查看父指标的波动是由

哪个子指标引发的。

(2) 指标拆解：可以对任何一个波动的指标进行多维分析，以缩小问题范围。

杜邦分析是一个典型的用在财务上的分析，但是它也可以用在性能领域，笔者在后面将会展示一个用于性能分析的公式。

杜邦分析能够帮助我们找到影响性能指标的子指标，在精准定位到是哪个指标的问题之后，我们还需要进行另外一种分析，即多维分析。

12.2.3 数据分析之多维分析

多维分析原本是一个专业的领域，传统的多维分析将观察事物的角度分成了多个维度，如图 12-6 所示。

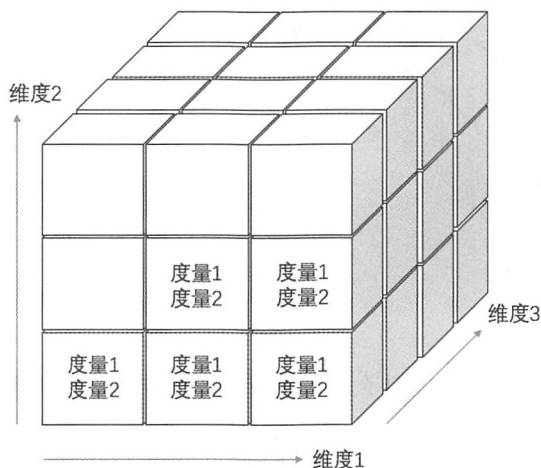


图 12-6

这里简单地介绍一个维度和度量的概念。

- 维度：观察数据的角度，比如在业务指标上，观察“最近 7 天交易金额”这个指标，可以从多个维度观察，比如人群标签维度、地区维度、商品品类维度等。
- 度量：度量某个具体的数据值，比如对最近 7 天交易金额进行求和操作就是一个度量。

下面用 SQL 来举个例子。

```
select sum(最近 7 天销售收入) from table_aa group by 省份, 产品类型
```

在这条语句中，省份和产品类型就是维度，sum（销售收入）就是度量。

这里的度量好像跟指标有些相似，它们的区别是：指标是面向业务的，度量是面向数据的。指标一般都是由度量和维度组成的，比如“华东区最近 7 日交易额”这个指标里面有地区维度，也有交易额汇总的度量。

多维模型在三维的情况下是可以用立方体来表示的，但是超过三维就必须用超立方体来表示，此时用图形表示超立方体是比较难理解的，所以笔者倾向于用二维模型来表示多维模型，如图 12-7 所示。

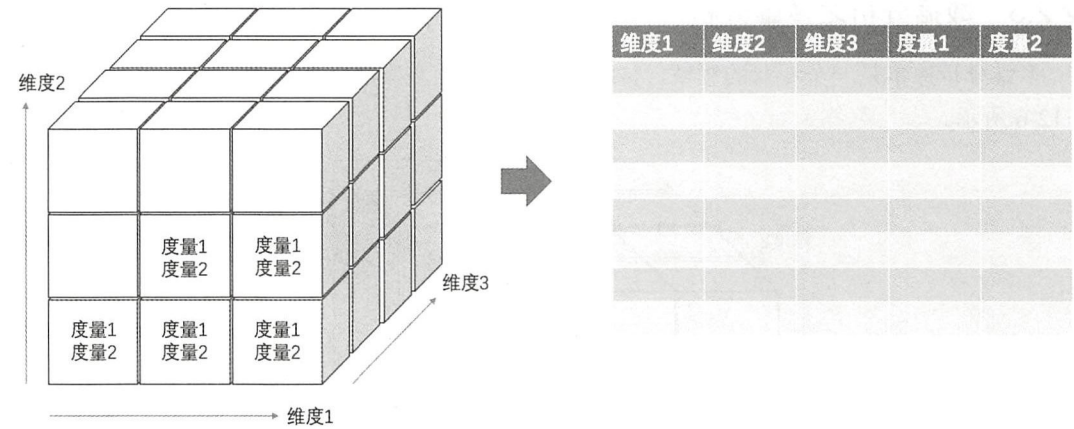


图 12-7

二维模型可以理解成一张表，用什么对表进行操作比较好呢？当然是用结构化查询语言（如 SQL）这样的查询方式来查表更加高效，学习曲线低、易上手。所以本章后续的描述将以维度模型为基础，以 SQL 作为分析手段来对性能问题进行诊断。

多维模型和二维模型的历史比较久远，在专业的 OLAP 相关的资料中会有一些介绍，本书不再更多地介绍它们的来龙去脉及未来趋势。

1. 分析技巧

当建立起维度模型之后，就需要使用这个多维模型了。

以前面的杜邦分析指标树举例，树上的每个指标都对应了一个维度模型，没有这个模型及其背后的数据就无法算出具体的指标值，所以假设维度模型如图 12-8 所示，其数据量达上万条。



| | A | B | C | D | E | F |
|----|----------|----|----|----|----|------|
| 1 | 日期 | 月份 | 城市 | 省份 | 品类 | 销售金额 |
| 2 | 2017/5/1 | 5 | 杭州 | 浙江 | 电视 | 5000 |
| 3 | 2017/5/1 | 5 | 绍兴 | 浙江 | 电视 | 1000 |
| 4 | 2017/5/1 | 5 | 宁波 | 浙江 | 电视 | 4000 |
| 5 | 2017/5/1 | 5 | 台州 | 浙江 | 电视 | 1000 |
| 6 | 2017/5/1 | 5 | 温州 | 浙江 | 电视 | 3000 |
| 7 | 2017/5/1 | 5 | 湖州 | 浙江 | 电视 | 2000 |
| 8 | 2017/5/1 | 5 | 嘉兴 | 浙江 | 电视 | 1000 |
| 9 | 2017/5/1 | 5 | 天台 | 浙江 | 电视 | 1000 |
| 10 | 2017/5/2 | 5 | 杭州 | 浙江 | 电视 | 5500 |
| 11 | 2017/5/2 | 5 | 绍兴 | 浙江 | 电视 | 1200 |
| 12 | 2017/5/2 | 5 | 宁波 | 浙江 | 电视 | 4500 |
| 13 | 2017/5/2 | 5 | 台州 | 浙江 | 电视 | 1200 |
| 14 | 2017/5/2 | 5 | 温州 | 浙江 | 电视 | 3500 |
| 15 | 2017/5/2 | 5 | 湖州 | 浙江 | 电视 | 2200 |
| 16 | 2017/5/2 | 5 | 嘉兴 | 浙江 | 电视 | 1100 |
| 17 | 2017/5/2 | 5 | 天台 | 浙江 | 电视 | 1300 |
| 18 | 2017/5/3 | 5 | 杭州 | 浙江 | 电视 | 4800 |
| 19 | 2017/5/3 | 5 | 绍兴 | 浙江 | 电视 | 900 |
| 20 | 2017/5/3 | 5 | 宁波 | 浙江 | 电视 | 3800 |
| 21 | 2017/5/3 | 5 | 台州 | 浙江 | 电视 | 700 |
| 22 | 2017/5/3 | 5 | 温州 | 浙江 | 电视 | 2500 |
| 23 | 2017/5/3 | 5 | 湖州 | 浙江 | 电视 | 1700 |
| 24 | 2017/5/3 | 5 | 嘉兴 | 浙江 | 电视 | 800 |
| 25 | 2017/5/3 | 5 | 天台 | 浙江 | 电视 | 800 |
| 26 | 2017/5/4 | 5 | 杭州 | 浙江 | 电视 | 3000 |
| 27 | 2017/5/4 | 5 | 绍兴 | 浙江 | 电视 | 600 |
| 28 | 2017/5/4 | 5 | 宁波 | 浙江 | 电视 | 2500 |
| 29 | 2017/5/4 | 5 | 台州 | 浙江 | 电视 | 600 |
| 30 | 2017/5/4 | 5 | 温州 | 浙江 | 电视 | 2000 |
| 31 | 2017/5/4 | 5 | 湖州 | 浙江 | 电视 | 1100 |
| 32 | 2017/5/4 | 5 | 嘉兴 | 浙江 | 电视 | 700 |
| 33 | 2017/5/4 | 5 | 天台 | 浙江 | 电视 | 700 |
| 34 | 2017/5/1 | 5 | 南京 | 江苏 | 电视 | 4500 |
| 35 | 2017/5/1 | 5 | 南通 | 江苏 | 电视 | 2000 |
| 36 | 2017/5/1 | 5 | 苏州 | 江苏 | 电视 | 4000 |
| 37 | 2017/5/1 | 5 | 常州 | 江苏 | 电视 | 2000 |
| 38 | 2017/5/1 | 5 | 无锡 | 江苏 | 电视 | 3000 |
| 39 | 2017/5/1 | 5 | 镇江 | 江苏 | 电视 | 1000 |
| 40 | 2017/5/1 | 5 | 扬州 | 江苏 | 电视 | 1500 |
| 41 | 2017/5/1 | 5 | 徐州 | 江苏 | 电视 | 2500 |

图 12-8

当销售额这个指标发生变化时，需要对指标进行拆解，而拆解的方法就是查看是哪个省份的哪个产品类型的销售额在时间维度上发生了什么样的变化：

```
select sum(销售收入), 产品类型, 日期 from table_aa group by 产品类型
```

日期查完之后，将数据通过可视化的效果展示出来，如图 12-9 所示。

得到了这份数据，可以知道大概是 4 号出了问题，但是不知道是哪个城市出了问题，于是加上城市维度：

```
select sum(销售收入), 产品类型, 月份 from table_aa group by 城市, 月份 where 产品类型 = `电视`
```

接下来通过一些可视化工具来对其进行可视化展现，以便更快地发现问题，如图 12-10 所示。



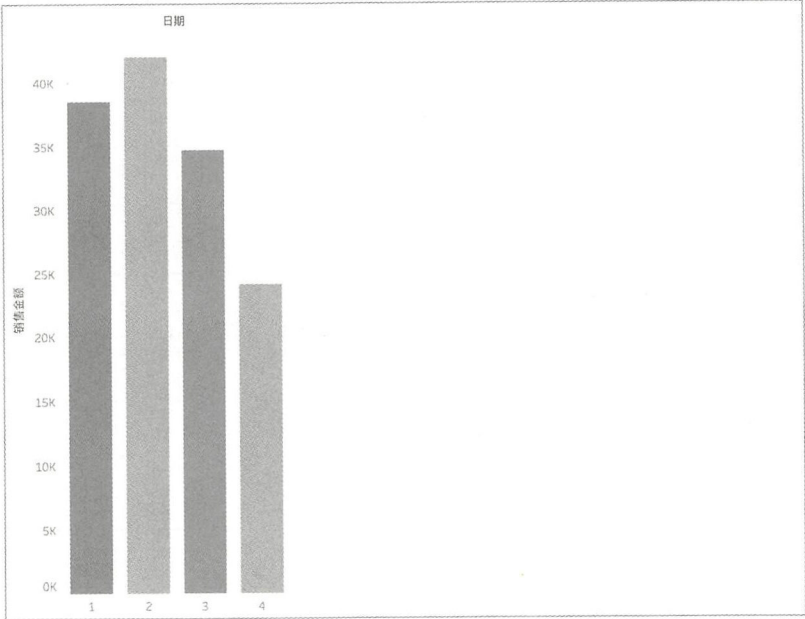


图 12-9

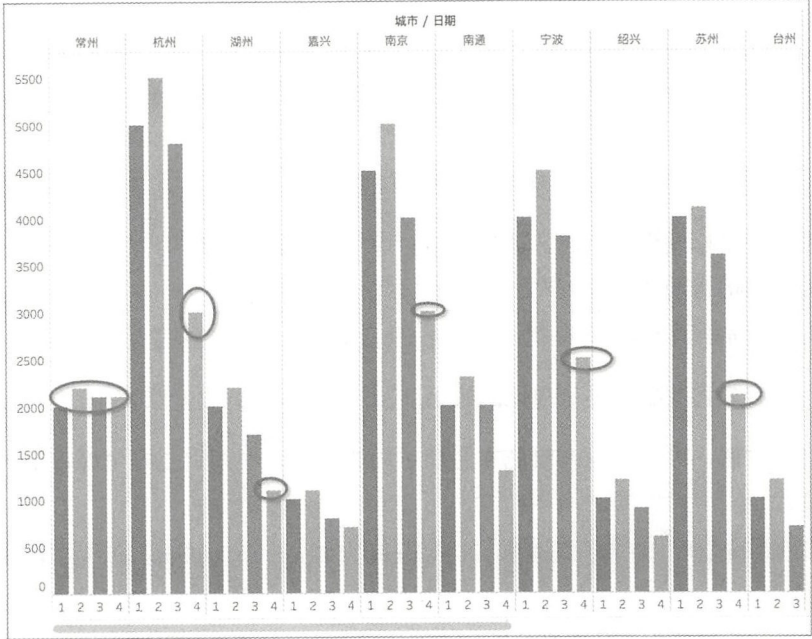


图 12-10



发现只有少数城市 4 号销售额没有下跌，大部分城市的销售额都下跌了。这好像也是说得通的，因为 5 月 4 号大家开始上班了，没有时间购物了。

但是常州在 4 号的销售额没有下跌，这是为什么呢？这要根据品类进行拆解，看看是不是有什么异常，比如其他品类的销售额下跌了，但是有一个品类的销售额上升了，我们就要详细调查为什么这个品类的销售额突然上升。

这就是通过多维分析的方式来缩小问题的范围的方法，它本质上适用于各行各业，在后面的章节中，笔者将会在性能优化的领域使用它。

2. 小结

以上简要地讲述了数据分析的相关概念和简单方法，总结如下。

- (1) 定义指标：如杜邦分析案例中的财务指标，会有很多个。
- (2) 确定指标之间的关联关系：指标有父子关系，父指标受到多个子指标的影响。
- (3) 采集数据：这里要进行埋点、采集，确保数据质量。
- (4) 维度建模：为每个指标建立维度模型，同时加工数据，映射成这个维度模型。
- (5) 指标追溯：通过每个指标的维度模型中的数据计算确定发生问题的子指标。
- (6) 指标多维分析：对子指标进行多维分析，找到发生问题的维度，缩小问题的范围，进而确定问题所在。

了解这个流程之后，来看一下如何用这套方法来解决 WebP 导致 L-D 转化率下降的问题。

12.3 通过数据分析来诊断 WebP 的性能问题

12.3.1 指标定义

在理解了杜邦分析之后，下面来数一数电商中经常出现的类似的顶层指标有哪些。

- (1) GMV：表示网站的交易额。
- (2) L-D 转化率：表示用户从 List 页面跳转到 Detail 页面的比例。
- (3) D-O 转化率：表示用户从 Detail 页面跳到下单成功页面的比例。
- (4) 客单价：每个客人的订单平均金额。



(5) 网站 UV：表示特定时间段内有多少消费者来到该网站。

一个网站的 GMV 路径很多，典型的有如图 12-11 所示的 3 条。

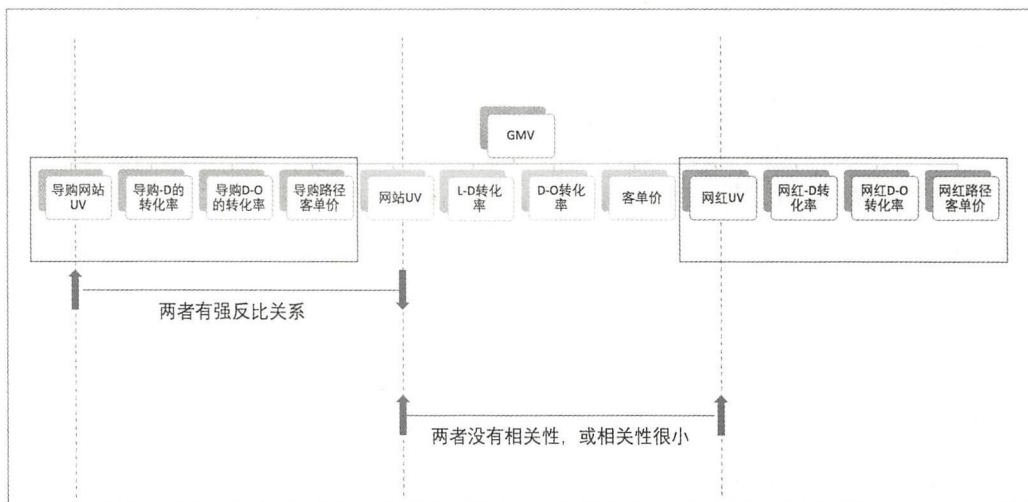


图 12-11

为了简化问题，只讨论主搜索这个转化漏斗，各指标之间的关系如下：

$$\text{GMV} = \text{网站 UV} \times \text{L-D 转化率} \times \text{D-O 转化率} \times \text{客单价}$$

这个函数关系是显而易见的，但是公式中的每个因子又是由哪些子因子组成的呢？这里以 L-D 转化率来重点阐述一下。

是什么导致一个用户有兴趣从 List 页面跳转到 Detail 页面进行商品详情的查看呢？可能有下面这几种因素。

(1) 搜索正确率：用户搜一个 iPhone 7，结果出来一堆 iPhone 7 的手机壳，这可能就不是用户想要的内容，用户自然就不会进入 Detail 页面。

(2) 价格：一个用户的消费能力是有级别的，对方想买一个 50~100 元的手机壳，但是系统给他推荐了 10 元的，自然他进 Detail 页面的可能性也降低了。

(3) 网页打开性能：面对一个需要 5 秒打开的搜索页面和一个需要 3 秒打开的页面，用户进入 Detail 页面的概率也是大大不同的。这里页面加载性能又分成如下两块。

① HTML 页面的下载和渲染速度。



② 图片的性能。如果 List 页面的图片打开速度特别快，且图片精美，那么也能影响 L-D 转化率。

(4) 营销活动：如果在 List 页面看到某些商品有优惠，那么用单击 Detail 页面的概率就增加了。

现在问题来了，之前在 GMV 和子指标之间有明确的函数关系，但是 L-D 转化率指标和其下的子指标存在什么函数关系呢？假设：

$$\text{L-D 转化率} = F(\text{搜索正确性指标, 价格指标, 性能指标, 营销活动指标})$$

我们知道，营销活动如何设定跟平台和卖家都有很大的关系，如果一个卖家出血本做营销活动，L-D 转化率一般都升高得比较明显，但是如果营销的优惠幅度下降，那么 L-D 转化率也会跟着下降。在这个场景中，平台和卖家投入的资金会非常影响 L-D。这是人为因素，这一人为因素将直接决定 L-D 转化率指标。只有人为因素产生了规律性，这个 F 函数才可能稳定下来。

12.3.2 基于指标树自动诊断 WebP 的性能问题

这里不观察子指标和父指标的函数关系，而是来观察当父指标波动时，机器是否可以快速诊断出是哪个子指标出现了问题。所以有必要搞清楚，父指标由哪些子指标构成。

接下来分析一下图片性能指标的子指标有哪些。

- (1) 图片大小。
- (2) 图片质量。
- (3) 用户下载图片的 RTT。
- (4) 边缘节点的缓存命中率。

根据上述分析，就可以构建指标树了，如图 12-12 所示。

这棵指标树是一个局部的展现，当 GMV 下降时：

- (1) 通过程序自动向下追溯，可以得知 L-D 转化率下降导致 GMV 下降。
- (2) 再往下追溯就会发现，图片性能下降导致了 L-D 转化率的下降。
- (3) 再往下追溯又发现，原来是由于节点缓存命中率的下降导致了图片性能的下降。



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

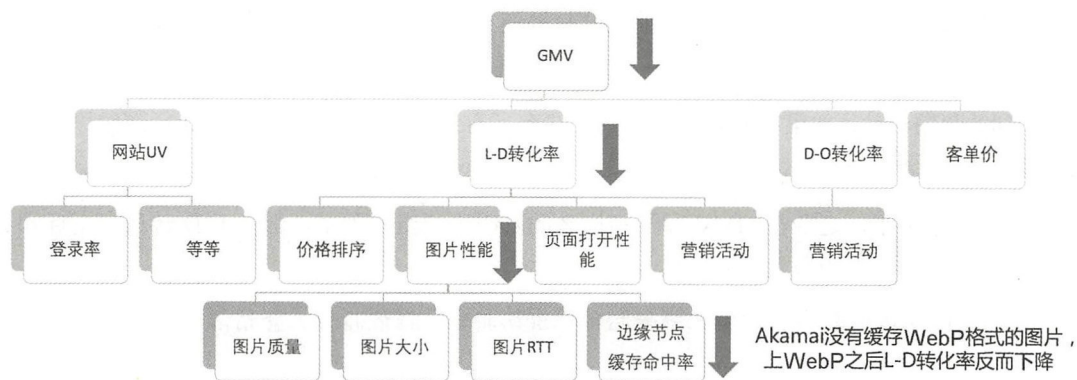


图 12-12

通过指标树向下追溯缩小了问题的范围。当问题缩小到边缘节点缓存命中率导致 L-D 转化率下降时，可以开展多维分析，再次缩小问题的范围。

在边缘节点的缓存命中率上，需要搜集数据，这份数据应该如图 12-13 所示，其中 RTT 可以简化管理为创建链接的时间，这两个时间相对其他时间来说较为接近。

| 国家 | 城市 | 图片大小 | 图片类型 | 终端 | TTFB(ms) | RTT(ms) |
|-----|------|------|------|--------|----------|---------|
| 法国 | 巴黎 | 70 | WebP | PC | 800 | 61 |
| 法国 | 马赛 | 85 | JPG | mobile | 60 | 40 |
| 法国 | 里昂 | 90 | JPG | mobile | 70 | 41 |
| 法国 | 巴黎 | 95 | JPG | mobile | 55 | 39 |
| 法国 | 马赛 | 60 | WebP | PC | 780 | 50 |
| 法国 | 里昂 | 87 | JPG | PC | 58 | 40 |
| 法国 | 巴黎 | 88 | JPG | mobile | 60 | 42 |
| 法国 | 马赛 | 60 | WebP | mobile | 850 | 65 |
| 法国 | 里昂 | 55 | WebP | mobile | 780 | 61 |
| 法国 | 巴黎 | 65 | WebP | PC | 790 | 61 |
| 法国 | 巴黎 | 89 | JPG | PC | 62 | 40 |
| 俄罗斯 | 莫斯科 | 70 | JPG | PC | 63 | 35 |
| 俄罗斯 | 莫斯科 | 88 | JPG | mobile | 55 | 31 |
| 俄罗斯 | 莫斯科 | 87 | JPG | mobile | 58 | 30 |
| 俄罗斯 | 圣彼得堡 | 90 | JPG | mobile | 59 | 30 |
| 俄罗斯 | 圣彼得堡 | 60 | WebP | PC | 810 | 59 |
| 俄罗斯 | 圣彼得堡 | 62 | WebP | mobile | 790 | 50 |
| 俄罗斯 | 圣彼得堡 | 61 | WebP | PC | 760 | 50 |
| 俄罗斯 | 莫斯科 | 50 | JPG | mobile | 60 | 38 |
| 俄罗斯 | 圣彼得堡 | 61 | JPG | PC | 65 | 40 |
| 俄罗斯 | 莫斯科 | 70 | JPG | PC | 58 | 35 |
| 俄罗斯 | 圣彼得堡 | 71 | JPG | mobile | 50 | 30 |
| 俄罗斯 | 莫斯科 | 75 | JPG | mobile | 55 | 40 |
| 俄罗斯 | 圣彼得堡 | 65 | JPG | PC | 61 | 42 |
| 俄罗斯 | 莫斯科 | 40 | WebP | PC | 810 | 45 |

图 12-13

应该通过什么样的分析技巧来发现这份数据隐藏的问题呢？多维分析的思路如下。



`select avg (边缘节点缓存命中率), 国家, 图片类型, 月份 from table_xx group by 国家, 图片类型`

分析结果如图 12-14 所示(这个分析结果的可视化对数据做了一些处理, 以便更好地理解)。

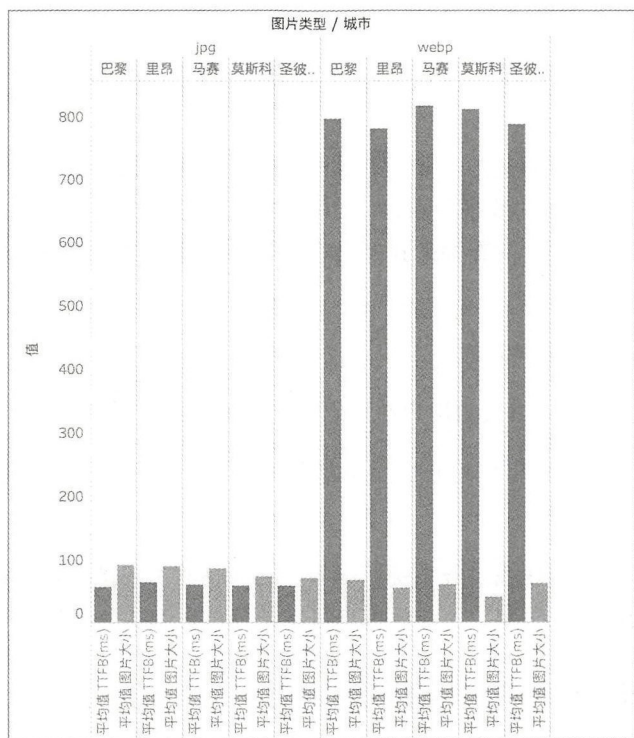


图 12-14

通过这样的分析, 应该马上就可以找出大小接近的图片, 但是 WebP 图片的 TTFB 高达 800ms, 远高于 JPG 图片。上述分析过程, 可以通过程序自动完成, 自然人要做的就是定义好指标树, 以及准备好对应的数据, 其他分析都可以通过机器来完成。机器每天发诊断报告, 供技术、运营、产品人员参考。

当然, 在实际生产环境中, 可能还有其他业务需求上线, 影响整体的 GMV, 从而导致向下追溯的时候很难判断到底是什么因素影响了 GMV。这个时候, 可以从第二层级来看指标波动的情况。比如 GMV 没变, 但是 UV 增加了, L-D 下降了, 这时候肯定有需求导致 UV 增加, 也有新增的内容导致 L-D 下降, 于是程序就从 L-D 往下追溯, 不能认为 GMV 没变就没什么问题。

通过类似杜邦分析的指标树拆解及多维分析, 甚至可以求得性能指标对整体 GMV 的影响,



关键在于当指标树的某个层次无法人为求得 F 函数的时候，要通过机器学习的方式得到父子指标之间的关系。如此一来，整体的指标函数就会变得完整，甚至可以事前预估性能优化的效果，以判断性能优化对 GMV 的影响。

12.4 案例：通过数据分析进行 OLAP 分析和 RT 优化

前面的例子是一个简单的案例，接下来看一个较为复杂的数据分析指导性能优化的案例。笔者工作的前 10 年，一直做在线事务系统，在线事务系统的一个典型特点是绝大多数的查询形式都是固定的。所以针对固定的查询形式，会产出各种各样的缓存或者调用方法来弥补高并发查询下的瓶颈。

在线分析系统有如下两个特点。

- (1) SQL 无规则：查询方式由用户在页面上拖曳的方式来决定，产生了五花八门的 SQL。
- (2) SQL 变化快：如果用户的分析行为发生变化或者产品发布新功能，查询的 SQL 就会发生变化。

这种系统的并发量会比在线事务系统低很多，它面临的主要挑战是，在各种各样的分析需求下，如何保障系统的响应时间，不能让用户每次做分析时都等几十秒。

12.4.1 在线分析系统响应指标基线的定义

1. 在线分析系统的响应时间过长问题

用户在查看报表或者进行数据分析的时候，每次都要等很长时间，有时候 RT 高达 90s，有时候 RT 是 1s。用户在做数据分析时不得不等待系统的返回，而这个时间片段又很难让用户转而去去做其他事情，这对用户来说会导致工作效率下降。

那么到底多长时间是一个比较合适的时间，这里有 Akamai 的一份调研数据。在 2006 年，一个来自 Akamai 的研究认为：通常 4s 左右的平均加载时间，可能是用户等待页面加载的最大时间。

这个 4s 的时间是如何计算出来的呢？这里有一个计算模型，里面包含了很多因子，比如行业水准、终端、网络、地域、转化率等，统一建模后得出了一个综合的结果。如果分领域，不同领域的值应该是不一样的。

在那个时间点，我们将这个指标定义为 OLAP 在 3s 内的比例为 95%。对应原来动辄十几秒



的查询时间，这已经有了质的飞跃了（当然从目前的水准看，这个指标完全可以定义成 1s 内 99% 的查询）。

2. 指标基线定义的方法和原则

1) 当前水准的判断

（1）跨行业参考：即使互联网用户的等待容忍时间是 4s（2006 年），也不代表在 OLAP 领域的等待容忍时间也是 4s。所以在这里定义的这个指标具有不确定性，因为无法确定是否可以达到这个目标。

（2）本行业指标基线建模：行业水准、终端、网络、地域、转化率等一系列因素，可以精准地计算出何种基线对产品是最有帮助的。

2) 即使 500ms 的查询时间是最优的，但是目前还达不到，所以指标基线的定义应该遵循以下原则：

（1）基线定义得太高，遥不可及会打击团队的信心和士气。

（2）基线定义得太低，毫不费力会让团队成长速度下降。

（3）合理的基线设定原则是全力以赴后可以达到，比全力以赴更努力可以超越既定基线。

所以，读者可以根据以上方法和原则设定自身领域的指标和指标基线。

12.4.2 性能问题诊断

在项目启动后，第一时间要确定两件事。

- 指标：查询在 3s 内的比例。
- 目标值：95%。

为了达到这个目标，接下来要对这个应用进行分析，简要诊断框架可以分成两个部分，一个是应用架构，另一个是系统流程。

首先要做的就是分析应用的构成，该应用包含的模块如图 12-15 所示。



图 12-15

大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

- OLAP Server: 接收用户请求，并决定将请求发送到哪里。
- Cache: 用来缓存用户查询过的数据。
- MR: SQL on Hadoop, 在 HDFS 上执行 Map-Reduce 任务来进行数据查询。
- queryengine: SQL on MPP 是一个基于列存的分布式查询引擎，目前 SQL on MPP 还有一些功能上的缺失，遇到这部分请求时，还是需要路由回 SQL on Hadoop。

有了系统的模块之后，下面来分析整个系统的主体调用流程，如图 12-16 所示。

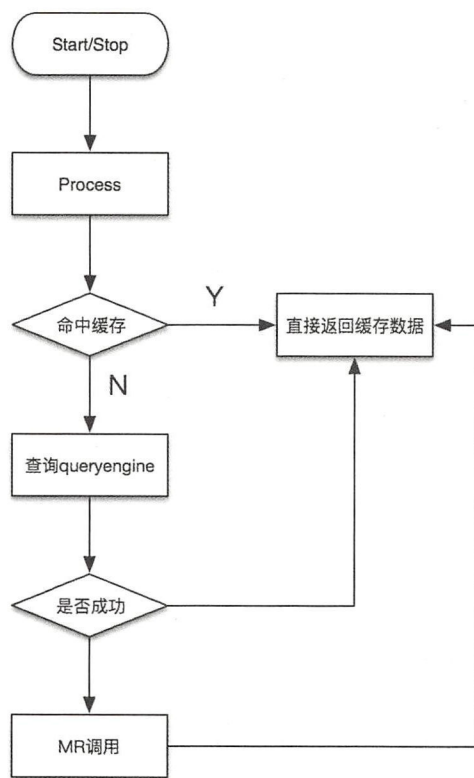


图 12-16

这是早期 OLAP 查询的一个具备共性的流程，这里存在一个很大问题，就是 MR 的调用耗时太长，经常高达 1min 以上，数据科学家很难忍受如此长的返回时间，要优化就必须先细化这个诊断框架。这里提出几个问题：

- (1) 缓存不能命中的比例是多少？
- (2) queryengine 的查询不能成功分为哪些情况，比例是多少？

(3) 这些不能成功的情况再细分有哪些情况，比例是多少？

这时需要获取对应的数据，通过对数据的观察来指导下一步的行动。

12.4.3 数据的获取及觉察

1. 数据建模

这里的维度如下。

- 查询模块：它有 3 个维值，分别为缓存、queryengine、MR。
- 查询来源：它的维值来自缓存失败、queryengine 失败（比如 MR 的查询可能是来自 queryengine 超时），以及直接用户查询（比如 MR 的查询可能直接来自用户，因为 queryengine 不支持同环比）。
- 查询时间：查询发生的时间。
- SQL：发生查询时具体的 SQL。
- 响应时间：查询的耗时。
- 是否成功：yes 或 no。
- 失败原因：它有几个维度，分别是超时、数据量过大、功能不支持（大表 join, count distinct）等。

2. 数据的获取

这里数据的获取也要通过打点，不过这个打点不是在浏览器上打点，因为分析的是后台查询流程，所以需要在后台代码中植入拦截器进行打点，然后通过日志的方式将其统一收集到大数据平台。

3. 数据分析

采集了上述数据之后，对其进行如下多维分析。

(1) 分析缓存查询所占的比例：

```
a = select count(*) / (select count(*) from table_xx) where 查询模块='缓存'
```

(2) 分析未走缓存时 queryengine 所占的比例：

```
b = 1 - a
```

① 分析查询走 queryengine 的比例：

```
select count(*) / (select count(*) from table_xx) where 查询模块='queryengine'
```

② 分析 queryengine 成功返回数据的比例：

```
select count(*)/(select count(*) from table_xx) where 查询模块='queryengine and 是否成功 = 'yes'
```

③ 分析走 queryengine 不成功的比例：

```
select count(*)/(select count(*) from table_xx) where 查询模块='queryengine and 是否成功 = 'no'
```

④ 分析走 queryengine 的响应在 3s 内的比例：

```
select count(*), SQL from table_xx where 查询模块='queryengine and 是否成功 = 'yes' and 响应时间 <= 3000
```

解读：3s 内的查询占比为 65.4%，计算公式如图 12-17 所示。3s 外的查询中 34%的请求落到 MR 上。对这份数据进行进一步多维分析，还发现如下细节。

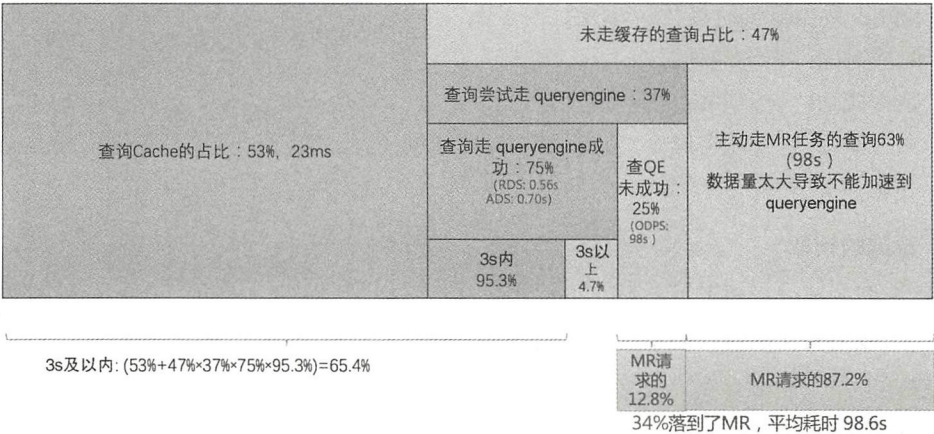


图 12-17

- (1) 缓存的查询占比相当高，达 53%。
- (2) 非缓存的查询占比分成了如下两部分。
 - ① 一部分走了 queryengine，47%中的 37%的 75%是走 queryengine 成功的。
 - ② 一部分走了 MR 任务：
 - 其中 47%中的 63%由于数据量太大导致不能走 queryengine。
 - 47%中的 37%的 25%走 queryengine 没有成功，最终走了 MR 任务。

根据这个结论，对走 MR 的查询请求数据进行再次分析，得到如图 12-18 所示的结果。

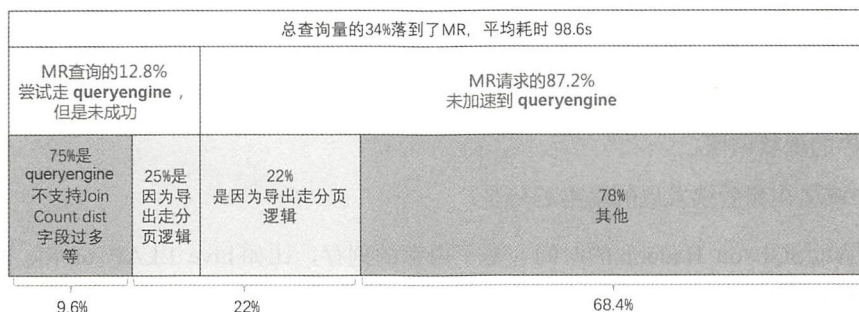


图 12-18

对图 12-18 进行解读可以得出解决方案：

- (1) queryengine 在 join、count distinct 等功能特性上有待增强，这些可以继续细化，确定 join 和 count distinct 等功能的比例。
- (2) 分页逻辑很多都落在了 MR 任务上，分页前的 count 都可以看作一个很大的消耗。
- (3) 其他的情况基本是数据量太大导致 queryengine 无法处理。

12.4.4 方案的推导

1. 基本的解决方案

根据这份数据如何才能达到目标？

- (1) 增强 queryengine 上的 join、count distinct 等功能。
 - ① 去掉 join，在 ETL 阶段将需要 join 的地方做成大宽表，从而将分析时的 join 去掉。
 - ② 通过 CBO 来优化 queryengine 上的 join 功能，可以参考 Spark 2.2 和 calcite 中的 CBO 优化算法。
 - ③ 通过 HRC 算法来做近似的 count distinct 操作，从而减少精确 count distinct 带来的开销。
- (2) 将精准的分页变成模糊的分页，去除每次分页时精准的 count 操作。
- (3) 将计算需要的数据量减少，提前做好数据的聚合再供分析引擎使用。

看上去可行，只要对这些内容再次进行细化，就可以评估出这 3 点具体能提升多少，要投入多少资源。

所以除了通过数据分析找到问题，还需要对其进行多维度思考。

(1) 业界维度：业界的相同或者相关领域目前在解决什么问题，是如何解决的。

(2) 业务场景维度：业务场景是什么样子的，和目前的解决方案是如何匹配的。

2. 业界的现状调研

这个领域现在和后续发展的基本趋势是：

(1) 缩短 SQL on Hadoop 的时间（基于内存的列存，比如 hive LLAP+orcfile 和 Spark 的 RDD+carbondata 之类的实现）。

(2) 完善“SQL on MPP+列存”的功能（各种 SQL 函数的支持、开窗、行级匹配、小计等）。

(3) 预计算，尤其是在报表的需求下，计算模式比较固定，可以使用预计算的各种框架，业界有不少此类产品。

虽然简单地写了几个方向，但是其中涉及的技术实在太多，仅一个 CBO 就可以研究很久，一个列存又要很多人研究很久，所以本章不会对使用这些技术的具体优化方法做详细介绍。

下面回到业务，来看看业务特征。

产品包含两个部分，一个是数据分析，另一个是制作好的报表，它们的特点如下。

(1) 数据分析：用户在产品中进行各种数据探查，选择不同维度、不同度量来探查可能发生问题的地方，用户的行为是完全不固定的。

(2) 报表：查询模式固定，维度和度量都已经定义好，查询条件也是固定的，如表 12-3 所示。

表 12-3

| | 多维分析场景（需求的占比为20%） | 报表场景（需求占比为80%） | 统计分析场景 |
|--------------------|------------------------------|---------------------|--------|
| SQL on Hadoop + 列存 | 适合 | 不适合 | 适合 |
| SQL on MPP + 列存 | 适合 | 不适合 | 不适合 |
| 预计算 | 不适合，如果全量计算cube所有组合，计算量大，浪费量大 | 非常适合，查询模式固定，预计算非常适合 | 不适合 |

将这个评估进一步细化，如图 12-19 所示。

通过这样的多维分析之后，基本就知道是应该把精力投在眼前的权宜之计上，还是投在值



得长期投资的事情上。

| | MPP | SQL on Hadoop (基于Spark和列存) |
|-------------|--|---|
| 元数据 | 自己管理 | HDFS的meta-data系统完成 |
| 资源隔离 | 自己管理 | Yarn |
| 对机器要求 | 需要相同, 水桶原理 | Yarn可以屏蔽计算资源的差异 |
| 集群规模 | 1000 (到达1000的规模已经要付出相当大的代价了) | 取决namenode的能力, 数千 |
| RT (同等资源下) | 小于MR, 秒级, 单次查询所有节点都需要参与运算, 并发度非常高, 基本就是全worker节点并发 | 也是秒级, 比MPP大。单次查询中的第一次查询需要从HDFS datanode中加载数据到多个worker节点内存 (Hive2.1的LLAP和Spark的dataframe), 第二次查询会直接使用内存中的数据并行计算。这个并行度取决于参与计算的worker节点数 (Yarn的调度) |
| QPS (同等资源下) | 比如MR高 | 比MPP高 |
| Failover机制 | 数据需要自己管理备份, 以便在某天机器发生问题时, 单次查询不会受到影响, 比如vertica的k-safy机制 | 某个节点挂了之后, 可以自动有其他节点接受task, 不会影响全局 |
| 可扩展性 | 加机器需要大量的数据重新平衡的过程 | HDFS自动完成 |
| 可维护性 | 对于MPP来说, 任何集群的改变都涉及拓扑结构的变更, 也可能会涉及不同机器之间数据的迁移, 因此当集群中机器数量多的时候, 依然维护复杂的数据管理模型会造成维护成本大幅度上升 | 简单, HDFS自动完成 |
| 函数集合 | 弱 | 强, 尤其支持统计函数 |
| 典型产品 | Hybridgp, vertica, impala, drill+parquet | Spark + carbondata, Hive2.1 + orcfile |

图 12-19

经过慎重的分析, 笔者决定: 先解决短期问题, 再解决长期问题。

短期问题经过逐个突破之后, 性能提升比较明显, RT 按时间变化的趋势如图 12-20 所示。

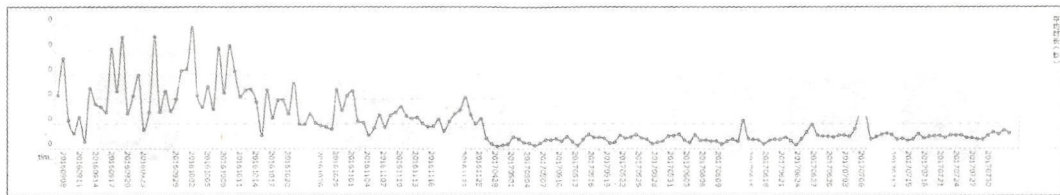


图 12-20

与此同时, 组建团队, 开始建设 M-OLAP。经过几个月的努力成功研发了智能 MOLAP 引擎, 在这个过程中, 同样用到了数据分析和统计学模型, 在每个场景精确地使用最合适的数据结构和算法。最终将普通报表查询的一秒查询性能指标提升到 90%。

12.4.5 小结

在这个案例中:

- (1) 定义了性能指标。
- (2) 根据对系统的熟悉程度来诊断系统中的每个关键环节或者模块的性能情况, 这里主要



做了打点采集数据的事情。

(3) 通过可视化效果，明确每个环节或者模块对性能的影响，然后针对具体影响性能的环节进行改进。

(4) 判断需要改进的点对整个架构的影响，从而判断改进的方案到底是权宜之计还是长远规划。

通过这样的数据分析方法性能得到了很大提升。

12.5 通过函数抽象进行性能优化

12.5.1 优化过程简介

在前面的优化中，通过简单的分组、聚合等基础数据分析技巧对性能问题进行了判断，这套方法能够将性能指标从 67% 提升到 90%。但是当笔者想把 1s 90% 的指标继续提升的时候，这套方法就比较吃力了，所以接下来讲一种更精细化、更体系化的方法，这个方法的一般过程如图 12-21 所示。

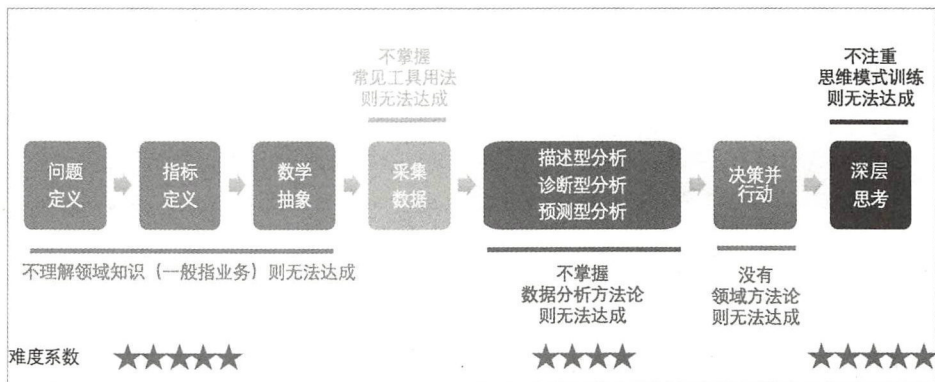


图 12-21

具体优化过程如下。

(1) 定义目标：目标是要达成的某种结果，这个结果一定是解决了某个问题的结果，可以用一个或者多个数字来衡量，如 Google 的 OKR 模型中的 objective。

(2) 指标定义：指标定义的前提是，要精准地把握待解决问题的本质，一般这个过程被称为问题定义，在这个前提下去定义指标、完成指标才能达到理想的效果。



- (3) 指标分解：构造数学模型，用数学模型来抽象架构和系统流程。
- (4) 打点采集数据：同时进行必要的清洗和加工。
- (5) 基础统计分析：缩小区间。
- (6) 多维分析：产出洞见。
- (7) 产出方案列表。
- (8) 深层思考：判定方案对架构的改进。
- (9) 使用决策树对指标树中的流量进行调度，以达到整体指标最高的目的。

12.5.2 函数抽象

在前面的案例中，获取了简化架构中的一个诊断框架，在本节中将尝试用数学模型来定义整个架构的性能，为了推导出这个公式，需要有一点简单的数据知识，还需要对整个系统的架构和模块有非常深入的理解，过程如下。

- (1) 指标定义，比如目标是一秒内查询占比大于 98%。
- (2) 梳理系统请求链路的每个关键环节。
- (3) 梳理关键环节中的每一个逻辑分支。
- (4) 根据请求链路中的每个关键环节和关键环节中的每个逻辑分支进行子指标拆解。
- (5) 将父指标和子指标进行函数抽象，如图 12-22 所示。

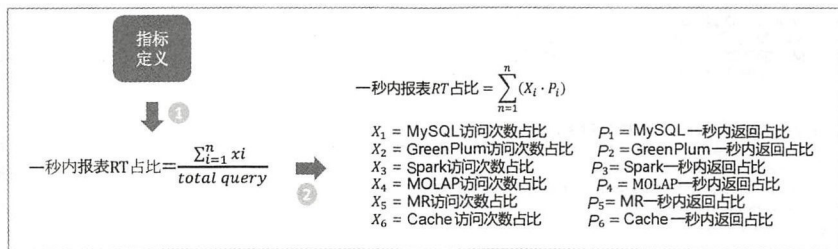


图 12-22

稍微解释一下这个公式：一个数据分析系统背后根据不同的业务场景对接了不同的计算引擎，有些是 Spark，有些是 GreenPlum，有些甚至是 MySQL，还有些是 MOLAP 引擎。不同引擎有不同特点，有些函数比较齐全，有些支持更大数据量的 join，有些性能极高，不同的场景



大型网站性能优化实战：从前端、网络、CDN 到后端、大促的全链路性能优化详解

使用不同的引擎，目前还没有一款引擎可以适合所有场景，尤其在大公司场景繁多的情况下，使用多种引擎协同工作是必要的。对于中小型公司，一款 MOLAP 加上一款 ROLAP，再加一个好的缓存系统，差不多就可以了，如图 12-23 所示。

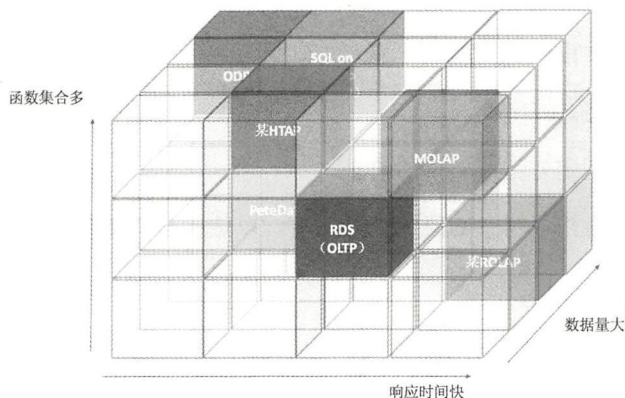


图 12-23

值得注意的是，这里 MySQL 一秒内返回的占比或者 GreenPlum 一秒内返回的占比只是这个链路上的占比，并不是引擎一秒内返回的占比，请求链路还有权限等模块会消耗性能，所以 MySQL 链路或者 GreenPlum 链路都指包含了权限和元数据操作等模块逻辑的链路。

在这个前提下，对这张图中公式的自变量再次进行展开，会得到如图 12-24 所示的结果。



图 12-24



这里比较简单，仅是对某个引擎链路访问占比的展开，比较复杂的是对某个引擎链路一秒内返回占比的展开，读者不用特别关心公式的具体含义，主要是明白其中的思路。

通过不断展开，就可以得到一个完整的函数，并不是所有的自变量都可以展开，有些就是一个统计量，还有一些自变量到后面很难手工展开，甚至要通过统计学的相关方法来得到自变量和其因变量的关系（比如有可能通过回归的方式得到线性或者非线性的方程等）。

得到这些自变量之后，要做的事情就是打点并采集数据，然后要做一些数据的清洗和加工工作，此处不详细展示这部分内容。

请注意，因为不能得出这样的函数模型，甚至连打点都是不精确的，所以这里用函数模型来表达系统中的某些关键指标是非常必要的。

12.5.3 统计分析

有了性能公式之后，接下来讲几个简单的概念。

1) 直方图

在直角坐标系中，横轴表示样本数据，纵轴表示频率与组距的比值，将频率分布表中各组频率的大小用相应矩形面积的大小来表示，由此画成的统计图叫作频率分布直方图（在图中，各个长方形的面积等于相应各组频率的数值，所有小矩形面积的和为 1）。

2) 组距，频数

全体样本分成的组的个数称为组数。每一组两个端点的差称为组距。落在不同小组中的数据个数为该组的频数。各组的频数之和等于这组数据的总数。频数与数据总数的比为频率（总频率等于各组频率之和，且它的值为 1）。频率大小反映了各组频数在数据总数中所占的分量。

在案例中，组距和频数的定义如下。

- 组距：500ms 为一个组距。
- 频数：任意 500ms 内发生查询的总数量。

接下来看一组以前的性能直方图数据（这里用折线图代替了直方图，数据经过了 ETL 清洗，调整组距是非常痛苦的事情），如图 12-25 所示。



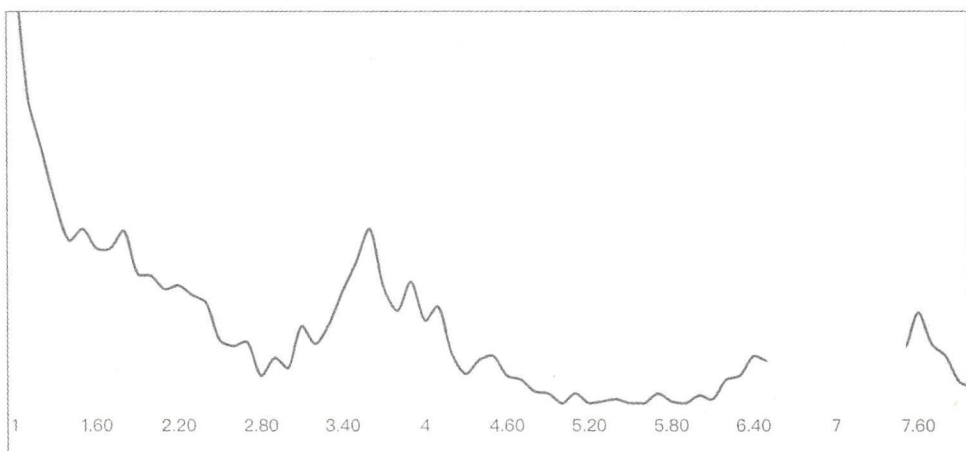


图 12-25

通过这个图形，立刻可以得知在 1s 外的请求的波动区间在什么地方，图中的时间范围是 1s 到 8s，实际上时间区间是可以选择的，时间在 30s 到 150s 之间都可以。在不停地优化过程中，新用户和场景出现时候，直方图的尖峰是在变化的，我们要做的就是找到直方图中波动的地方，进行多维分析，找到尖峰出现的原因。

此时多维分析的数据格式如下：

| 表名 | SQL 类型 | RT | 访问时间 |
|-----|--------|----|------|
| axx | | | |
| bxx | | | |
| cxx | | | |

比如图 12-25 中显示在 2.8s 到 4.6s 之间有个尖峰，我们要将之削掉。

```
select 表名 (or SQL 类型) count(*) from table group by 表名 (or SQL 类型) where RT >=2.8 and RT <=4.6
```

通过这样的数据分析，可以找到到底是什么表或者什么类型的 SQL 导致了在 2.8s 到 4.6s 之间的这个尖峰。

但是如果数据非常大，比如有几万条，此时通过多维分析可能找不到什么线索，怎么办呢？

可以缩小直方图的组距，原来是 500ms 一组，可以将其缩小至 100ms 一组，再来看会有新的发现。这可以看作一种缩小粒度的方式。



当然也可以反其道而行之，放大组距也有可能看到更宏观的问题。

12.5.4 小结

做工程的人员，对面向对象的抽象和架构的抽象都是比较熟悉的，但是在数据分析领域，利用函数来抽象整个系统是最重要的，这里需要的具体技巧和方法都不一样。读者可以有意识地培养自己数据分析方面的抽象思维。通过这个案例的介绍，可以了解通过数据分析来推动性能优化的流程，其中的关键步骤如图 12-26 所示。

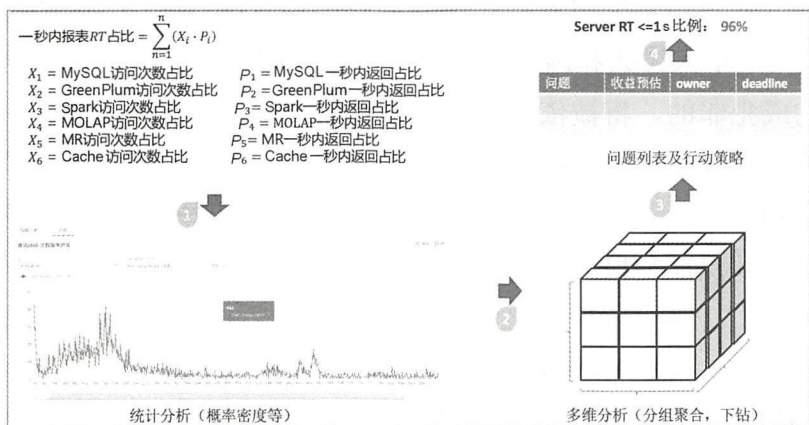


图 12-26

统计分析在发现问题方面是非常重要的，只从多维分析角度来做数据分析是不够的，还要通过正态分布和回归做性能的异常告警等，在此不做详细的阐述。

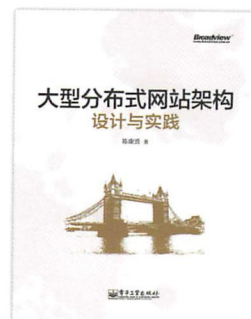
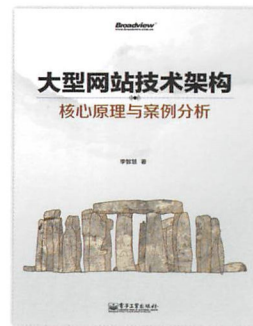
由于大数据时代的到来，通过数据分析和机器学习来改进工程指标已经越来越普遍，所以如果想要在性能领域有更深的造诣，通过数据分析和机器学习来改进性能是不可缺少的技能。

在数据分析中还有一件事比较重要，就是数字敏感性，这往往决定了分析者是否一眼可以看出问题，数据领域的老手一般都具备比较强的数字敏感性，随着大数据时代的到来，做工程的人员也需要慢慢培养这样的感觉。

本书历时有点长，大部分章节是挤出碎片时间进行总结的，难免有不合理的地方，希望读者可以批评指正。正如在序言里说到的，本书的特色是全链路优化的体系，大部分是针对实战过程做的总结，希望对读者的日常工作有帮助。



好书分享



大型网站性能优化实战

从前端、网络、CDN到后端、大促的全链路性能优化详解

作者在阿里近十年的技术架构实践中，总结出了一套体系化的大型网站性能优化方法论，又经过一年多时间方成此书。这个性能优化的技术体系是端到端的完整性能解决方案，可直接用于指导PV十亿级网站的性能优化，可帮助技术团队建立全局性能分析、监控和调优方案，可实现用较小的技术成本换得更好的系统性能。本书对于电商网站架构规划、社交网站性能调优、移动互联网和物联网通信架构的性能优化都有实际的参考价值。作者本人与我在阿里共事近八年，一起经历了一个网站从小到大的过程，其全面的技术分析视角，深入协议层的技术攻关能力，对于技术创造商业价值的思考，都可以在书中感受到。

· 阿里资深总监 叶军（不穷） ·

大型分布式网站在高速发展的过程中，整体系统容量和用户体验的性能调优至关重要，笔者以在阿里负责全球速卖通网站性能优化架构演进的亲身经历，从Web前端到服务端，从外部链路到内部机房，沉淀了大量的全链路性能问题分析思路和实战解决方案，非常值得一阅！

· 蚂蚁金服资深技术专家 雷继斌（雷舍） ·

涛明是我认识多年的合作伙伴，经常发现他对细节寻根刨底，对于大型网站架构、网站性能优化、CDN的使用，都可以做到从底层到上层全局把控。如果你想真正把网站性能消耗的来龙去脉理清楚，本书是非常好的从入门到精通的书籍，建议应用开发工程师都来读一下。

· 阿里云CDN资深技术专家 文景 ·



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview

上架建议：大型网站

ISBN 978-7-121-35002-3



9 787121 350023 >

定价：79.00元



责任编辑：董 英
封面设计：李 玲